Calibración y evaluación de cores RISC-V en entornos de simulación y FPGAs
Calibration of RISC-V cores in simulation and FPGA environments: evaluation and analysis

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Rubén de Mora Losada

Directores
Daniel Ángel Chaver Martínez
Fernando Castro Rodríguez

Doble Grado en Ingeniería Informática - Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Calibración y evaluación de cores RISC-V en entornos de simulación y FPGAs
Calibration of RISC-V cores in simulation and FPGA environments: evaluation and analysis

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Rubén de Mora Losada

Directores
Daniel Ángel Chaver Martínez
Fernando Castro Rodríguez

Convocatoria: ***Junio*** **2025**
Calificación: *10 (SB)*

Doble Grado en Ingeniería Informática - Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

2 de septiembre de 2025

Dedicatoria

A mi hermano, a mi madre y a mi padre, por haber estado, estar y porque siempre estarán a mi lado

Agradecimientos

Gracias a mi familia. Solo ellos saben lo duro que ha sido este camino y soy consciente de que en algunos momentos han sufrido tanto la carrera como yo. Nunca podré devolveros lo que me dais a diario; un camino duro se hace menos duro si vas con la mejor compañía, y yo la he tenido.

Gracias a mi pana. Nunca podré pensar en la carrera sin acordarme de ti.

Gracias a mis tutores, Dani y Fernando. Gracias por vuestro tiempo y por la ayuda que me habéis dado en todo momento. He tenido la mejor guía posible.

Gracias a mis amigos del colegio, instituto y universidad, siento no haber quedado tanto con vosotros durante estos años... ¡Aquí está el resultado!

Resumen

Este trabajo presenta la evaluación y calibración de un núcleo de procesador RISC-V mediante el refinamiento de su comportamiento en simulación con respecto a su ejecución real en una FPGA. El proyecto se centra en el núcleo CVW-Wally, un procesador de 64 bits de código abierto diseñado para investigación y docencia. Wally ha sido sintetizado y volcado en una FPGA Digilent Arty A7 y evaluado mediante el conjunto de benchmarks PolyBench. En paralelo, también se desplegó el núcleo CVA6 en una placa Genesys 2.

La plataforma de simulación gem5 se ha utilizado para modelar la microarquitectura en orden de Wally. Se ha desarrollado una configuración personalizada basada en MinorCPU que, iterativamente, se ha ido ajustando para obtener el comportamiento más parecido posible al observado en la FPGA. Este proceso ha implicado la configuración de jerarquías de memoria, predictores de saltos, anchos de pipeline y latencias de unidades funcionales, entre otros parámetros. En ambos entornos, se han recopilado métricas de rendimiento, incluyendo instrucciones ejecutadas y ciclos para calibrar el modelo de simulación.

Los resultados muestran que la configuración final de gem5 logra una alineación precisa con el comportamiento del hardware de Wally en términos de CPI y recuento de instrucciones, a lo largo de múltiples benchmarks y tipos de datos. Esto valida el modelo simulado y lo convierte en una herramienta fiable para futuras exploraciones arquitectónicas.

Palabras clave

RISC-V, Wally, CVA6, gem5, FPGA, calibración, contadores hardware, PolyBench

Abstract

This work presents the evaluation and calibration of a RISC-V processor core by aligning its behavior in simulation with that observed on a real FPGA implementation. The project focuses on the CVW-Wally core, a 64-bit open-source processor designed for research and educational use. Wally was deployed on a Digilent Arty A7 FPGA and benchmarked using the PolyBench suite. In parallel, the CVA6 core was also deployed on a Genesys 2 FPGA board.

The gem5 simulation platform was used to model Wally's in-order microarchitecture. A custom configuration based on MinorCPU was iteratively tuned to reproduce the behavior observed on the FPGA. This process involved configuring memory hierarchies, branch predictors, pipeline widths, functional unit latencies, and more. Performance metrics, including executed instructions and cycles, were collected in both environments to calibrate the simulation model.

Results show that the final gem5 configuration achieves strong alignment with the Wally hardware in terms of CPI and instruction behavior across multiple benchmarks and data types. This validates the simulation model and provides a reliable platform for further architectural exploration.

Keywords

RISC-V, Wally, CVA6, gem5, FPGA, calibration, performance counters, simulation, PolyBench

Contents

List of figures

List of tables

Chapter 1

Introduction

1.1. Motivation

RISC-V has emerged as a leading open-source instruction set architecture (ISA) in both academic and industrial domains. Its modular design and permissive licensing enables a high degree of flexibility in developing, customizing, and evaluating processor cores. As the ecosystem continues to grow, the need for reliable simulation tools becomes increasingly important, particularly for validating new microarchitectures and analyzing performance.

Within this landscape, multiple open-source RISC-V cores have been developed to serve different design goals. CVA6 is a processor designed to support complex software stacks, while CVW-Wally is a simpler in-order core aimed at education and architectural research. Although these cores differ in complexity, both are valuable for understanding RISC-V systems.

This project focuses on CVW-Wally and aims to calibrate its behavior in simulation using gem5, a widely adopted architectural simulator. To support comparison and reference, CVA6 was also deployed on the Genesys 2 FPGA platform. The core objective is to ensure that gem5 accurately replicates Wally's behavior as observed on its physical implementation on the Arty A7 FPGA, making the simulator a reliable tool for future architectural exploration and performance evaluation.

1.2. Objectives

The main goal of this work is to evaluate and calibrate the behavior of the Wally RISC-V core by comparing its physical execution on an FPGA with a cycle-level gem5 simulation. Specifically, the objectives include:

- Deploy the CVW-Wally core on the Arty A7 FPGA platform under a Linux-based environment.

- Run a set of representative benchmarks (PolyBench) and measure performance metrics using hardware counters.
- Recreate the core's behavior in gem5 using a customized simulation model based on MinorCPU.
- Iteratively calibrate the simulation to match hardware metrics in terms of instruction count, cycle count, and CPI.
- Deploy CVA6 on a Genesys 2 FPGA board as a comparative reference for architectural exploration.
- Deliver a calibrated gem5 configuration usable for research, performance validation, and system-level design studies.

1.3. Work Plan

The project was carried out over several months as part of a larger research effort within a project in parternship with IMEC[1], where I was working under contract. Weekly meetings were held with the team to coordinate progress, identify issues, and share results.

Additionally, I was awarded a research internship funded by IMEC, which will allow me to spend three months abroad continuing this work. During this time, I will help extend and refine the simulation framework, incorporating new benchmarks and validating additional microarchitectural components.

This undergraduate thesis will eventually serve as the foundation for a more complete research publication once the work is fully extended, validated, and documented.

The work plan followed a progressive structure, as shown in Figure 1.1. The first two months were dedicated to getting familiar with gem5, an architecture simulator I had not previously used. This phase focused on understanding its configuration system, architecture models, and successfully running initial simulations. Once this foundation was established, I began acquiring and preparing the necessary hardware and toolchains for deploying the Wally core on the Arty A7 FPGA. In parallel, I also completed the setup of the CVA6 core on the Genesys 2 board.

Following this, I replicated the deployment process for Wally, which was faster thanks to the experience gained with CVA6. With both hardware environments running Linux, I proceeded to develop a gem5 simulation model of Wally. The remainder of the project was spent calibrating this model, tuning parameters to match hardware behavior, and finally, running benchmarks and collecting performance measurements to validate the results.

[1] `https://www.imec-int.com/en`

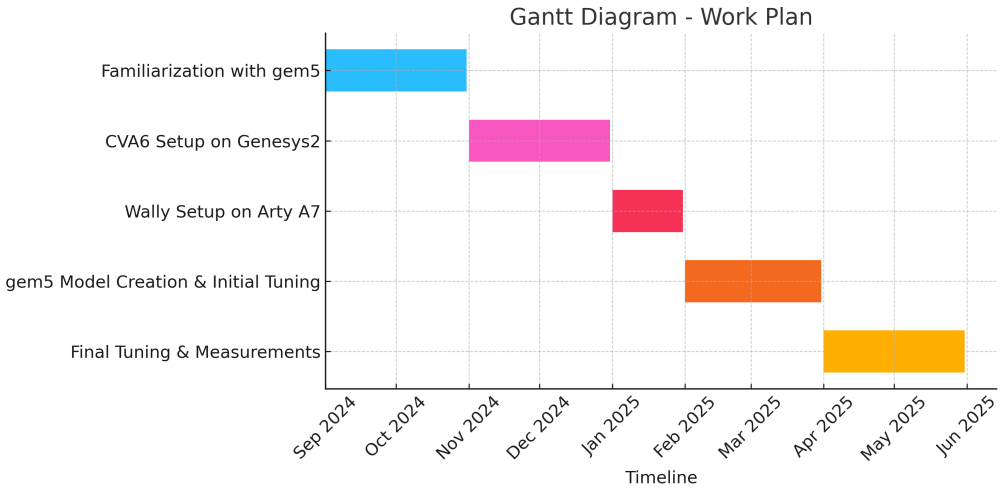


Figure 1.1: Gantt diagram – Work plan for the project timeline.

Chapter 2

State of the Art

This chapter provides a review of the current state of the art relevant to this work, focusing on three key areas.

First, it introduces the RISC-V instruction set architecture, highlighting its main features and the reasons behind its growing adoption in both industry and academia.

Second, it provides an overview of the current RISC-V ecosystem, examining both commercial vendors and existing RISC-V cores. It discusses major contributors to the industry, as well as prominent open-source and commercial cores, highlighting their design goals, architectural features, and typical application domains.

Finally, it presents some existing work on calibration and evaluation applied to RISC-V cores, which is the area most directly related to the research developed in this project.

2.1. RISC-V

RISC-V is a modern instruction set architecture (ISA) that has significantly impacted the microprocessor development landscape. Introduced by the University of California, Berkeley in 2010 (Waterman et al., 2011), it has since attracted substantial interest from both academia and industry due to its simplicity and improvements over traditional ISAs.

In contrast to proprietary ISAs such as X86 or ARM, RISC-V is released under an open-source license, which means it is free to use, implement, and modify without any fees. This has led to rapid global adoption, promoting a collaborative ecosystem in which universities, start-ups, and large corporations contribute to its development and optimization.

RISC stands for Reduced Instruction Set Computer. It is considered reduced in contrast to CISC (Complex Instruction Set Computer) architectures, such as Intel's x86 CPUs. The term reduced refers to the smaller number of instructions and the simpler design of each instruction existing in the instruction set. In CISC architec-

tures, individual instructions can perform multiple complex operations. In contrast, RISC architectures use simpler instructions that perform a single operation. As a result, while more instructions may be needed to accomplish a task in RISC compared to CISC, the instructions themselves are easier to execute.

This simplification also results in more predictable performance and less complex hardware design, which is particularly advantageous in modern computing environments where energy efficiency, scalability, and cost-effectiveness are critical. These attributes make RISC-V especially well-suited for a wide range of applications, from low-power embedded systems to high-performance computing and artificial intelligence workloads.

RISC-V defines a modular ISA composed of a small base instruction set and multiple optional extensions that cover various functional areas. Tables 2.1 and 2.2 present an overview of the base ISAs and standard extensions, respectively.

RISC-V also defines a canonical order in which these extension letters must be listed when naming an ISA subset: `I M A F D G Q L C B J T P V N`. For example, `RV32IALCBN` is valid, while `RV32IFADC` is not.

2.2. RISC-V Cores

As described above, RISC-V has gained considerable popularity not only in the academic world, but also within the commercial sector. Its open and flexible instruction set architecture has opened the door to a wide range of innovations. Unlike proprietary ISAs, RISC-V allows developers to build custom processors without licensing restrictions, which has led to a growing number of startups dedicated exclusively to RISC-V development. At the same time, many well-established semiconductor companies have started to incorporate RISC-V cores into their products, recognizing the advantages of openness, adaptability, and long-term sustainability that the RISC-V ecosystem offers.

2.2.1. Commercial Vendors

Several companies have become key players in the commercial RISC-V market, offering a wide range of processor cores designed to meet diverse application requirements:

- **SiFive**[1] is the most prominent company in the RISC-V ecosystem. Founded by the original creators of the RISC-V ISA, SiFive offers a comprehensive portfolio of cores organized into distinct families: *Essential*, targeting microcontrollers and IoT devices; *Performance*, delivering high compute density for modern workloads such as mobile and datacenter environments; *Intelligence*, optimized for AI and machine learning applications; and *Automotive*, which focuses on functional safety and reliability for automotive-grade systems .

[1] `https://www.sifive.com`

Table 2.1: RISC-V Base ISAs

Name	**Description**
RV32I	Base Integer Instruction Set – 32-bit
RV32E	Embedded Base Integer Set – 32-bit, 16 registers
RV64I	Base Integer Instruction Set – 64-bit
RV128I	Base Integer Instruction Set – 128-bit

Table 2.2: RISC-V Standard Extensions

Name	**Description**
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
G	Shorthand for IMAFD
Q	Quad-Precision Floating-Point
L	Decimal Floating-Point
C	Compressed Instructions
B	Bit Manipulation
J	Dynamically Translated Languages
T	Transactional Memory
P	Packed-SIMD Instructions
V	Vector Operations
N	User-Level Interrupts
H	Hypervisor Instructions
S	Supervisor-Level Instructions

- **Andes Technology**[2] provides a wide variety of RISC-V processors, ranging from low-power embedded cores to high-performance vector processors. Their cores are widely adopted in automotive electronics, artificial intelligence accelerators, networking, and IoT applications. Andes is also a key contributor to RISC-V ISA standardization.

- **Codasip**[3] specializes in customizable RISC-V cores that emphasize extensibility and efficiency. Through their unique processor design automation tools, Codasip enables SoC designers to create application-specific processors by extending the base RISC-V ISA with domain-specific instructions, one of the key strengths of the RISC-V ecosystem.

- **Openchip**[4] is a Spanish company that focuses on the development of energy-efficient RISC-V processors and SoCs. With an emphasis on open hardware and academic collaboration, Openchip provides core IPs suitable for both commercial and research applications. Openchip has been selected by the European Commission as an *Important Project of Common European Interest* to support research, innovation, and industrial deployment of microelectronics and communications technologies across the entire value chain.

2.2.2. Notable RISC-V Cores

The RISC-V ecosystem includes a wide variety of core implementations, ranging from simple embedded designs to complex, high-performance processors. The following is a selection of notable open-source cores that have made significant contributions to the RISC-V ecosystem.

- **Rocket Core**[5]: Developed at UC Berkeley, Rocket is one of the earliest and most widely used 64-bit in-order RISC-V cores. It is part of the Rocket Chip SoC generator and serves as a reference implementation for many academic projects and commercial prototypes.

- **BOOM (Berkeley Out-of-Order Machine)**[6]: Also developed by the UC Berkeley team, BOOM is a superscalar, out-of-order RISC-V core aimed at high-performance applications. It is frequently used in research related to processor microarchitecture, speculation, and instruction scheduling.

- **VeeR Cores**: Developed by Western Digital, the VeeR core family targets embedded and edge computing with energy-efficient RISC-V implementations. The *EH1*[7] is a 32-bit, 2-way superscalar, in-order processor featuring a 9-stage

[2] https://www.andestechnology.com/
[3] https://codasip.com/
[4] https://openchip.com/
[5] https://chipyard.readthedocs.io/en/main/Generators/Rocket.html
[6] https://boom-core.org/
[7] https://github.com/chipsalliance/Cores-VeeR-EH1

pipeline. In contrast, the *EL2*[8] is a lightweight 32-bit core designed for ultra-low power consumption. It implements a simpler 4-stage in-order pipeline and supports only single-issue execution. Although it lacks a Memory Management Unit (MMU), it provides Physical Memory Protection (PMP) and supports Machine mode, with optional support for User mode in certain configurations.

- **XiangShan**[9]: An open-source high-performance out-of-order RISC-V core developed by the Institute of Computing Technology at the Chinese Academy of Sciences. XiangShan is notable for its advanced microarchitecture and scalability, aiming to match commercial performance cores.

In the following sections, we focus specifically on **CVA6**, due to its openness and widespread use in academic research, and **Wally**, a newly released open-source core that presents a promising platform for future development and experimentation.

Wally[10]**:** Core-V-Wally (CVW) is an open-source 64-bit RISC-V core developed at Harvey Mudd College (California), specifically designed as a versatile platform for educational purposes and research exploration. It employs a five-stage pipeline architecture, illustrated in Figure 2.1, with clearly defined stages: Fetch, Decode, Execute, Memory, and Writeback.

In the Fetch stage, the Wally core fetches instructions from an instruction cache, utilizing branch prediction mechanisms to minimize pipeline stalls. The Decode stage is responsible for interpreting the fetched instructions and includes a decompression module to handle compressed instructions. Following this, the Execute stage incorporates integer execution units (IEU), a multiply-divide unit (MDU), and a floating-point unit (FPU), enabling comprehensive arithmetic and logic operations.

The Memory stage handles data access operations and includes support for virtual memory with a memory management unit (MMU) and hardware page table walker (PTW). Data interactions are conducted via a dedicated data cache, which ensures efficient memory management and fast data retrieval. The Writeback stage finalizes instruction execution by writing results back to the appropriate registers.

Additionally, Wally incorporates robust support for privileged operations, control/status register (CSR) management. Wally's modular architecture enables straightforward customization of essential features, including the memory system, cache hierarchies, and memory management unit (MMU). This flexibility makes it an excellent candidate for experimental setups, facilitating targeted architectural modifications and rigorous evaluations.

Due to its clear, modular, and well-documented design, Wally significantly simplifies the integration of instrumentation required for performance calibration and detailed evaluation studies. According to the official OpenHW Group repository[11],

[8] `https://github.com/chipsalliance/Cores-VeeR-EL2`
[9] `https://github.com/OpenXiangShan/XiangShan`
[10] `https://github.com/openhwgroup/cvw`
[11] `https://github.com/openhwgroup/core-v-cores`

Wally is targeted at Education and will be accompanied by an engineering textbook and course on computer architecture. In this project, we had early access to this textbook, which provided valuable guidance during the development and analysis phases. For this particular project, a configuration of Wally implementing the complete RV64GC ISA extension set was utilized. This configuration includes comprehensive support for integer operations, atomic instructions, compressed instructions, and both single and double-precision floating-point operations, thus covering a broad spectrum of application needs.

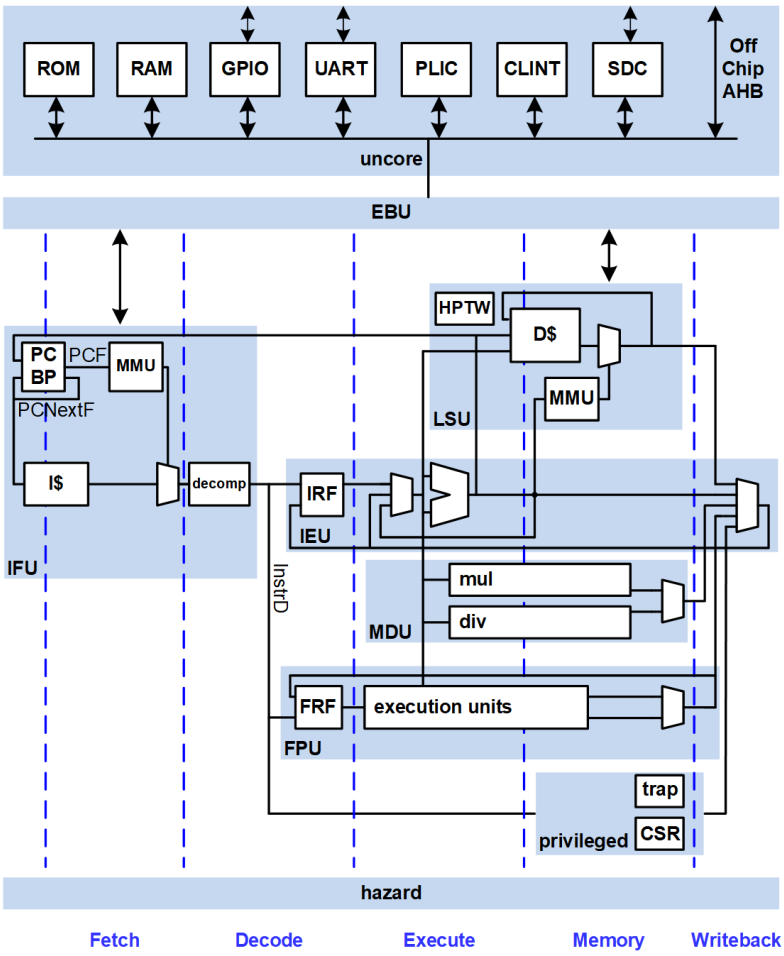


Figure 2.1: Wally Pipeline

CVA6[12]**:** CVA6 (formerly known as Ariane) is a 64-bit application-class RISC-V core developed by the Parallel Systems Architecture group at ETH Zurich and currently maintained by the OpenHW Group. It employs a six-stage, single-issue, in-order pipeline architecture, depicted in Figure 2.2, and fully implements the RV64GC instruction set.

The pipeline structure of CVA6 consists of six clearly defined stages: Instruction Fetch, Instruction Decode, Execute, Memory Access, Writeback, and Commit.

[12] `https://github.com/openhwgroup/cva6`

CVA6 supports three privilege levels: Machine (M), Supervisor (S), and User (U), which enable the execution of Unix-like operating systems such as Linux. A key capability of CVA6 is its comprehensive support for virtual memory management, featuring dedicated hardware page table walkers (PTW) and separate Translation Lookaside Buffers (TLBs) for instructions and data. This robust memory management infrastructure facilitates efficient address translation and contributes significantly to overall system performance.

To enhance control flow efficiency, the core integrates fundamental branch prediction mechanisms, including a branch target buffer (BTB) and a branch history table (BHT).

Memory interactions in CVA6 are conducted through AXI4 interfaces for both instruction and data accesses, ensuring compatibility and flexibility in various system-on-chip (SoC) configurations. Additionally, CVA6's design is highly parameterizable, offering options to modify cache configurations, pipeline features, and MMU settings, thereby making it a versatile platform for research, prototyping, and SoC development.

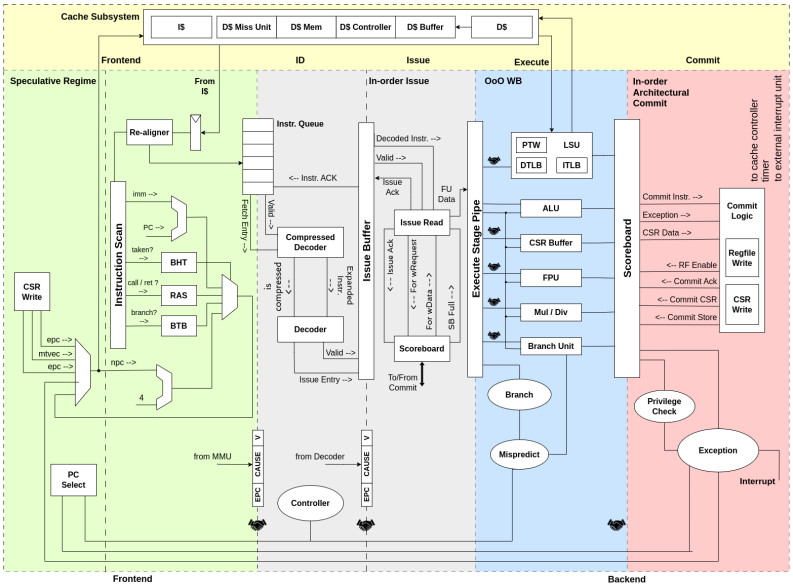


Figure 2.2: CVA6 Pipeline

2.3. Calibration of RISC-V Cores

The calibration of RISC-V cores is a crucial step in ensuring their correctness, performance, and energy efficiency. Calibration typically involves adjusting internal simulator parameters to minimize discrepancies between simulated behavior and the real hardware. In this work, this process is guided by measurements obtained from FPGA implementations: by comparing the performance of selected benchmarks on both platforms, simulation parameters are tuned to make the model closer to the

actual hardware. Several previous works have focused on the calibration of RISC-V cores in different contexts.

Several recent works have explored the calibration and validation of RISC-V systems, particularly focusing on the CVA6 core and the gem5 simulation framework. These efforts aim to bridge the gap between simulated and real hardware performance, ensuring reliable microarchitectural analysis.

A number of studies leverage gem5 to evaluate and optimize CVA6. In Ravenel et al. (2023), the authors present a CVA6-based simulation infrastructure in gem5, enabling early-stage architectural exploration and pathfinding. Similarly, Allart et al. (2024) proposes enhancements to CVA6 by implementing a superscalar version based on a detailed performance model. Additional insights into the CVA6 SoC architecture are discussed in Shahid et al. (2024), where a gem5-based platform is used to analyze design trade-offs and bottlenecks.

Simulation accuracy is a critical concern in full-system modeling. The work in Kioulos (2023) performs a preliminary evaluation of the accuracy of simulated RISC-V systems by comparing gem5 outputs with real hardware performance. Complementarily, Pai et al. (2023) explores the use of SimPoints to reduce simulation time while maintaining representative behavior, validated against physical boards. A more systematic approach is presented in Pathak et al. (2023), where a validation framework is proposed for full-system simulators targeting RISC-V platforms.

Performance monitoring has also been the subject of several studies. In Domingos et al. (2021) and Domingos et al. (2023), the authors investigate how to support RISC-V performance counters through existing Linux tools such as ‘perf‘, facilitating performance profiling and calibration from the OS level.

Lastly, memory hierarchy calibration has been addressed in Huppert et al. (2021), where the authors focus on tuning gem5 parameters to match the behavior of real in-order RISC-V cores. This approach enhances the credibility of simulated memory performance metrics and is crucial for workloads sensitive to cache and memory latency.

Chapter 3

Experimental Environment

This section serves as an introduction to the platforms used in the evaluation and calibration of the CVW-Wally RISC-V core. On the hardware side, the CVW-Wally core was synthesized and deployed on the Digilent Arty A7 FPGA board. On the simulation side, the `gem5` architectural simulator was used to create a configurable model of Wally's in-order microarchitecture. Although the CVA6 core was also deployed on the Genesys 2 FPGA board for exploratory and reference purposes, the calibration process focused exclusively on CVW-Wally.

To drive the evaluation, we used benchmarks from the PolyBench suite, which offer a variety of compute-intensive kernels across several domains. These benchmarks were executed on both hardware and simulation platforms to analyze execution time, number of instructions, and cycle counts.

All simulation and calibration experiments were executed on a virtual machine hosted within the ArTeCS (Architecture and Technology Computer Science) Group from UCM.

The following section presents the hardware platforms (Genesys 2 and Arty A7), the gem5 simulator, the benchmark suite and the virtual machine.

3.1. Arty A7

The Arty A7 board[1], depicted in Figure 3.1, is a versatile and ready-to-use development platform centered around Xilinx's Artix-7 FPGA (part numbers XC7A35TICSG324-1L and XC7A100TCSG324-1 for the A7-100 variant). Designed specifically for applications involving the MicroBlaze Soft Processing System, the Arty A7 offers flexibility, adapting effectively to a wide range of project requirements.

Key specifications of the Arty A7 include FPGA logic composed of 5,200 logic slices (15,850 logic slices in the A7-100 variant), 1,800 Kbits of Block RAM (4,860

[1] `https://digilent.com/reference/programmable-logic/arty-a7/start?srsltid=AfmBOoo_RgiV6RcHWAVg0bW-lmnPNySkbQrRxejxw0u1w1ES-x_7BrkW`

Kbits in the A7-100 variant), and 90 DSP slices (240 in the A7-100 variant). It features 256 MB of DDR3 memory running at 333 MHz (667 MT/s), an internal clock frequency of over 450 MHz, and 16 MB Quad-SPI flash memory. The board supports Ethernet connectivity at 10/100 Mbps and includes an integrated on-chip analog-to-digital converter.

Connectivity and onboard peripherals of the Arty A7 include four Pmod connectors, an Arduino/chipKIT-compatible shield connector, four user switches, four user buttons, a reset LED, four user LEDs, and four RGB user LEDs. This diverse set of features allows the Arty A7 to serve as a robust platform for applications ranging from communication systems equipped with UART, SPI, I2C interfaces, and Ethernet MAC, to precise timing and control tasks with numerous timers.

The board supports programming via JTAG and Quad-SPI flash, providing versatile configuration options suitable for educational, prototyping, and experimental research projects.

The Wally core was deployed on physical hardware using the Arty A7 board. Following the setup provided in the official cvw repository, the configuration files for the `rv64gc` variant were used to synthesize and generate the FPGA bitstream. This bitstream was then used to program the Arty A7 board via Vivado 2020.1.

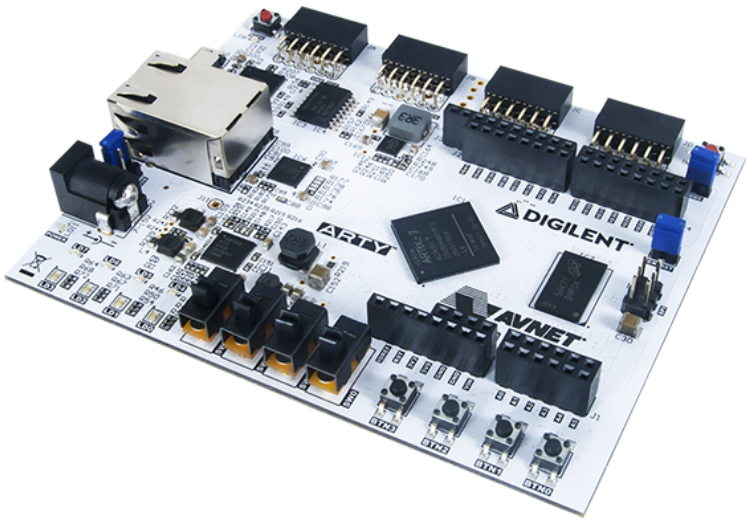


Figure 3.1: ArtyA7 board

3.2. Genesys2

The Digilent Genesys 2 board[2], depicted in Figure 3.2, is an advanced, high-performance digital circuit development platform based on Xilinx's Kintex-7 FPGA (part number XC7K325T-2FFG900C). It provides a robust environment ideal for data and video processing applications due to its high-capacity FPGA, fast external memories, and multiple high-speed digital interfaces.

Key specifications of the Genesys 2 FPGA board include FPGA logic consisting of 50,950 logic slices (each containing four 6-input LUTs and eight flip-flops), 16 Mbits of Block RAM, 840 DSP slices, and 10 clock management tiles (CMTs) each equipped with PLLs. The board supports DDR3 memory operating at 1800 MT/s with a 32-bit data width and an internal clock frequency exceeding 450 MHz. It also features a 16 MB Quad-SPI flash memory, a 10/100/1000 Ethernet PHY, and an on-chip analog-to-digital converter (XADC).

Connectivity options and onboard peripherals include five Pmod connectors, VGA and HDMI connectors (supporting both HDMI sink and source), a microSD card slot, USB interfaces for HID devices such as mouse and keyboard, USB mass storage support, and an audio codec with four 3.5mm jacks. Additionally, it offers a fully-populated 400-pin FMC HPC connector equipped with ten GTX lanes, eight onboard switches, and six buttons.

The Genesys 2 board supports programming through JTAG and Quad-SPI Flash, providing flexible configuration options suitable for diverse applications. This comprehensive feature set and extensive connectivity options make the Genesys 2 FPGA development board a versatile platform for educational, research, and prototyping purposes.

The CVA6 core was deployed on physical hardware using the Genesys 2 FPGA development board. Following the guidelines provided by the official OpenHW Group repository[3], the bitstream was generated and used to program the FPGA using Vivado 2018.2. The Genesys 2 setup includes a DDR3 memory controller, UART, SPI interface for SD card access, basic GPIOs connected to LEDs, and a boot ROM containing a zero-stage bootloader and device tree.

Programming was carried out by flashing the `ariane_xilinx.mcs` file onto the Spansion SPI flash memory using Vivado's hardware manager. Once the bitstream was successfully loaded, the system was booted directly from the configuration memory. After startup, UART communication was established to interact with the running core, verifying that CVA6 was functioning correctly on the FPGA. This hardware-based setup provided valuable hands-on experience and a platform for real-world observation of the core's behavior outside of simulation.

[2] `https://digilent.com/reference/programmable-logic/genesys-2/start?srsltid=AfmBOooqhGCfMeG0aWP_N8q0uD3VoOl3BCezqZSiG29gwc69aqRAnumw`

[3] `https://github.com/openhwgroup/cva6`

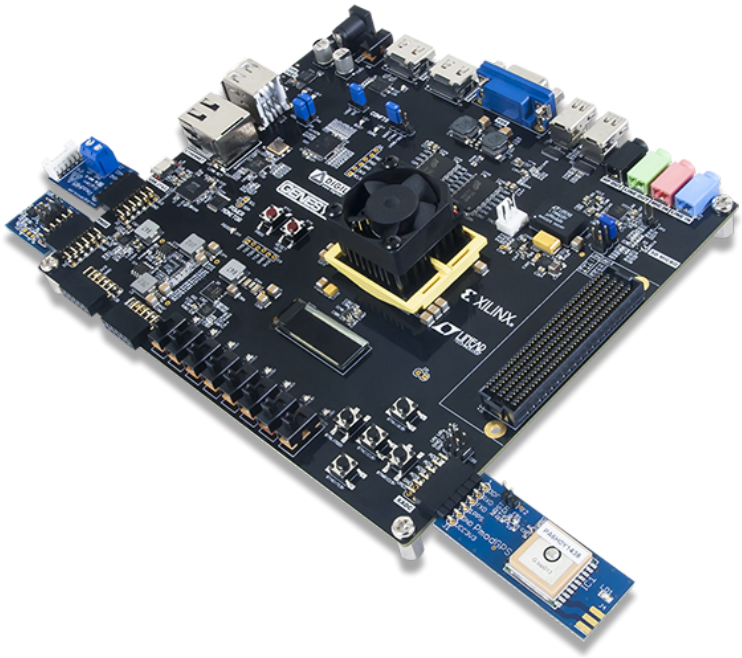


Figure 3.2: Genesys 2 board

3.3. Gem5 simulator

`Gem5`[4] is a highly flexible and widely used open-source architectural simulator with over 15 years of development. It is primarily employed in the research and development of computer systems and processor microarchitectures. The simulator supports accurate modeling of a wide range of hardware architectures—from simple uniprocessor systems to complex multi-core and multi-die configurations. Furthermore, gem5 supports multiple instruction set architectures (ISAs), including x86, ARM, and RISC-V, which makes it a versatile tool for simulating a variety of modern computing environments.

Gem5 offers two main simulation modes: Full System (FS) and Syscall Emulation (SE). In FS mode, the simulator emulates an entire system, including the hardware and the operating system. This enables the booting of a full Linux distribution, allowing users to execute basic commands and run full-stack applications. FS mode behaves similarly to a virtual machine, providing a realistic and detailed simulation environment. On the other hand, SE mode skips the OS boot process and instead emulates system calls directly. It emphasizes core and memory system simulation, making it significantly faster, albeit less accurate than FS. SE mode is suitable when the goal is to measure performance without needing the full operating system, while FS mode is preferred when realistic behavior and interaction with the OS are required.

Internally, gem5 combines Python, C++, and compiled binaries to define and

[4] `https://www.gem5.org/`

execute simulations. Python scripts serve as the main configuration interface, allowing users to define system parameters such as CPU type, memory hierarchy, and interconnects. These high-level scripts abstract away the complexity of the underlying C++ implementation, making it easier to describe different architectural setups. The actual simulation engine and hardware models are written in C++, which provides the necessary performance and detail to model components like processors, buses, caches, and interconnects. Finally, the code that runs inside the simulated architecture—whether in FS or SE mode—consists of compiled binaries, usually benchmarks, specifically built for the target ISA.

3.4. Benchmarks

To analyze and compare the performance of the core, the PolyBench benchmark suite has been selected. PolyBench is widely used in the computer architecture research community due to its well-structured, static control flow kernels and portability across simulation and hardware environments.

PolyBench includes a diverse set of benchmarks that fall into four main categories:

- **Data Mining**: workloads focused on statistical analysis and pattern detection in large datasets.
- **Linear Algebra**: classic matrix and vector operations commonly used in scientific computing.
- **Medley**: a mixed group of kernels that don't fall into the other categories, but are still representative of real workloads.
- **Stencils**: iterative solvers and simulations based on partial differential equations that operate on regular grids over time.

Each benchmark can be used with different datasets inputs of different sizes: `mini`, `small`, `medium`, `large`, and `extra large`. This flexibility allows experiments to be adjusted to the computational capabilities of the target platform. Moreover, benchmarks are available in three data types: `int`, `float`, and `double`, enabling analysis across different levels of numerical precision.

3.5. ArTeCS Virtual Machine

All simulation and calibration experiments were conducted on a virtual machine hosted within the ArTeCS (Architecture and Technology Computer Science) Group from UCM. This virtual machine runs on a Haswell-based server and is equipped with the following specifications:

- **Processor**: Intel Xeon E5-2695 v3 (launched in 2014)

- **Memory**: 65GB RAM
- **Architecture**: 14-core, 28-thread (2.30 GHz base frequency)

The virtualized environment provided sufficient resources to run full-system simulations with gem5, execute benchmarks, and perform calibration tasks efficiently. Additional details on the processor can be found at the official Intel product page[5].

3.6. Verilator

To inspect the microarchitectural behavior of the Wally core at cycle level, we used Verilator to simulate execution traces from hand-written RISC-V assembly programs. This setup allows observing how instructions are processed in the pipeline and how functional units (such as the ALU) behave across clock cycles.

The workflow followed is documented in a GitHub project[6]. The first step is to configure the environment variables and toolchain paths accordingly.

```
export VERILATOR_ROOT=/home/rvfpga/verilator_5-032/
export PATH=$VERILATOR_ROOT/bin:$PATH
export INCLUDE_PATH=$VERILATOR_ROOT/include

export RISCV=/home/rvfpga/RiscvToolchain/
export PATH=$RISCV/bin:$PATH

export WALLY=/home/rvfpga/Wally/cvw/
export PATH=$WALLY/bin:$PATH
```

To enable waveform generation, we modified the Verilator simulation Makefile located at `$WALLY/sim/verilator/Makefile` by adding the following flag:

```
PARAMS?=--trace
```

The following commands compile, simulate, and visualize the execution of an example loop written in assembly:

```
cd ~/Wally/cvw/examples/asm/example\_extended
riscv64-unknown-elf-gcc -march=rv64im -mabi=lp64 -nostdlib
```

[5] https://www.intel.com/content/www/us/en/products/sku/81057/intel-xeon-processor-e52695-v3-35m-cache-2-30-ghz/specifications.html

[6] https://github.com/artecs-group/RVfpga-sim-addons/tree/main/RVfpga_Microcredential/Module3/OtherRiscvCores

```
-o example\_extended example_extended.S
riscv64-unknown-elf-objdump -d example_extended > example_extended.objdump
wsim --vcd --sim verilator rv64gc --elf example_extended
gtkwave ~/Wally/cvw/sim/verilator/testbench.vcd
```

An example of an assembly code is shown in Figure 3.3. This program increments a register in a loop until it matches a constant value:

```
00000000000100b0 <rvtest_entry_point>:
   100b0:       02a00513                li      a0,42
   100b4:       00000393                li      t2,0
   100b8:       24000813                li      a6,576

00000000000100bc <REPEAT>:
   100bc:       00138393                addi    t2,t2,1
   100c0:       ff039ee3                bne     t2,a6,100bc <REPEAT>

00000000000100c4 <self_loop>:
   100c4:       0000006f                j       100c4 <self_loop>
```

Figure 3.3: Example of assembly code executed in Verilator.

Figure 3.4 shows the output waveform of the Verilator simulation. It includes the clock signal, the 32-bit instruction value, and the corresponding ALU result for each cycle.

This type of simulation provides direct visibility into the pipeline stages and instruction-level timing, and it was particularly useful for measuring functional unit latencies and verifying pipeline behavior during calibration.

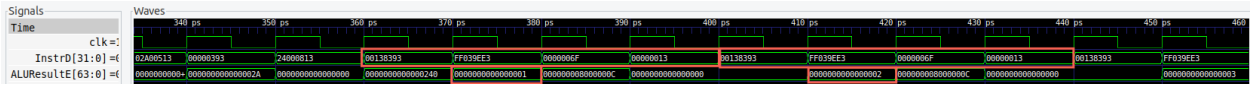


Figure 3.4: Cycle-level waveform of the loop execution using Verilator. The ALU result shows the incremented `t2` register value in each iteration.

Chapter 4

Methodology

This chapter describes the complete experimental setup and methodology used to calibrate and evaluate the performance of the Wally and CVA6 RISC-V cores, both on physical hardware (FPGAs) and in simulation (gem5). The primary objective is to align the behavior of simulated models with that of the real hardware by collecting performance metrics such as cycles and instruction counts, and then iteratively refining simulation parameters.

We begin by detailing the FPGA setup, including the necessary hardware components, toolchains, and configuration steps for both the Wally core on the Arty A7 board and the CVA6 core on the Genesys 2 platform. This includes the process of preparing bootable Linux images, generating FPGA bitstreams, and connecting to the target system via UART to run and monitor benchmarks.

Next, we describe the gem5 simulation environment, focusing on the Full System (FS) mode used to replicate the software environment found on the FPGA. We explain how a custom RISC-V image builder was created to streamline the setup process and ensure consistency across platforms.

The chapter also explains the compilation for the PolyBench suite to target our RISC-V architecture. We outline the benchmark configurations, data types, and dataset sizes used throughout the experiments.

Finally, we detail the calibration process, comparing the performance of the Wally core on FPGA and in gem5. This includes a discussion of the measurement methodology in both environments, the iterative tuning of architectural parameters in the simulator, and the use of auxiliary tools such as Verilator to extract detailed information about the microarchitecure, such as functional unit latencies or memory latency. This methodology enables a high-fidelity simulation model that closely matches the behavior of the target hardware.

4.1. FPGAs Setup

4.1.1. Wally & Arty A7

To evaluate the CVW-Wally core on physical hardware, we deployed it on the Digilent Arty A7 FPGA board. This setup was guided by the official documentation provided in the CVW[1] GitHub repository and was validated using Vivado 2019.1.

The deployment of Wally requires the following hardware components: a Digilent Arty A7 FPGA board, an SD card along with a compatible SD card reader, a USB A to Micro-B cable for programming and UART communication, and a Pmod SD card adapter to physically connect the SD card to the FPGA board.

The full process involves four main stages. First, a bootable Linux image must be built and flashed onto the SD card, including any benchmarking workloads or custom files. Second, the FPGA bitstream must be generated from the CVW source files using Vivado's build system. Third, the resulting bitstream is programmed onto the FPGA through Vivado's Hardware Manager. Finally, a terminal must be connected to the board via the UART interface to observe and interact with the Linux system running on the Wally core.

The following sections describe each of these steps in detail.

4.1.1.1. Preparing the SD Card

In order to run Linux on the Wally system, the SD card must be flashed with a compatible Linux image. This process is managed through the cvw repository, which provides all the necessary build scripts, root filesystem contents, and kernel configuration files. The resulting image enables the Wally core to boot a full Linux environment from the SD card once the system is powered on.

This setup was carried out from a Windows Subsystem for Linux (WSL) environment, which requires additional steps to expose USB devices such as SD card readers. By default, WSL does not recognize these devices natively, so the following procedure must be followed to make the SD card reader visible within WSL.

Note: The same procedure may be followed to make other USB-connected devices visible in WSL, such as FPGAs for Vivado programming.

Making the SD Card Reader Visible in WSL Begin by connecting the SD card reader (with the SD card inserted) and open a WSL terminal. In parallel, launch a PowerShell window with Administrator privileges and run the following command to list connected USB devices:

```
usbipd list
```

[1] `https://github.com/openhwgroup/cvw`

Locate your SD card reader in the list. A practical way to identify it is by unplugging and replugging the device while re-running the command. Once identified, note its associated `busid` (e.g., `1-19`).

Then, bind the device using:

```
usbipd bind --busid=<busid>
```

Next, attach the bound device to your WSL instance:

```
usbipd attach --wsl --busid=<busid>
```

Return to your WSL terminal and verify that the SD card is now accessible using:

```
sudo fdisk -l
```

You should now see your SD card listed as a device, typically something like `/dev/sde`. An example of this procedure is shown in Figure 4.1, which illustrates the use of the commands to make the sdcard visible (name sde).

To unbind and remove the device later, you may use:

```
usbipd detach --busid=<busid>
usbipd unbind --busid=<busid>
```

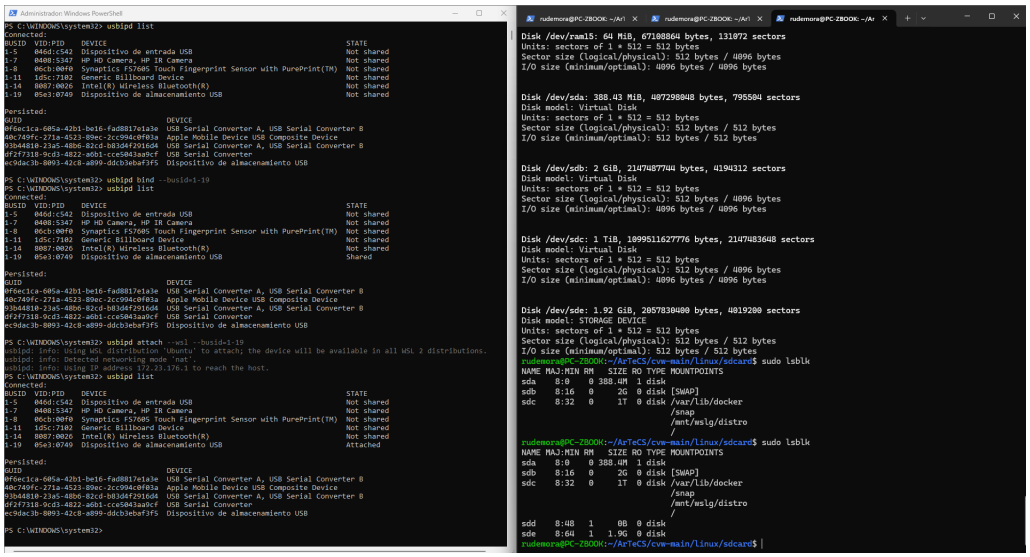


Figure 4.1: Connecting an SD card reader to WSL using `usbipd` and confirming visibility with `lsblk`.

Flashing the Linux Image Once the SD card reader is accessible from WSL, the next step is to flash the Linux image onto the card. First, identify the correct device path using:

```
lsblk
```

or

```
fdisk -l
```

Before flashing, it is recommended to clean and format the SD card to ensure no conflicting partitions remain. This can be done by wiping the partition table with the following command:

```
sudo dd if=/dev/zero of=/dev/sd<letter> bs=1M count=10
```

This step clears the first few megabytes of the card, removing any existing filesystem or partition information.

First, the CVW repository must be cloned from GitHub using the following command:

```
git clone https://github.com/openhwgroup/cvw
```

Navigate to the flashing script directory and run:

```
cd linux/sdcard
./flash-sd.sh -d wally-artya7.dtb /dev/sd<letter>
```

As shown in Figure 4.2, this script creates a GPT (GUID Partition Table) and sets up four partitions: partition 1 contains the device tree binary, partition 2 stores OpenSBI, partition 3 includes the Buildroot Linux system, and partition 4 provides a writable ext4 filesystem.

After the script completes, you can safely remove the reader and the SD card.

Adding Benchmarks or Custom Files to the SD Card Once the Linux image has been successfully flashed onto the SD card, you may want to include custom files or benchmarking binaries for later use on the target system. After inserting the SD card into your computer, run:

```
sudo fdisk -l
```

This command will list the partitions on the SD card. If you see at least one partition labeled as a Linux filesystem, you can use that partition directly to copy your files (in that case, skip ahead to mounting the partition). Otherwise, or if you prefer to create a new partition specifically for this purpose, follow these steps:

```
GPT Information for /dev/sde ====================================
Disk /dev/sde: 4019200 sectors, 1.9 GiB
Model: STORAGE DEVICE
Sector size (logical/physical): 512/512 bytes
Disk identifier (GUID): E7C1A05A-A425-467E-BDB1-26F1B16BF230
Partition table holds up to 128 entries
Main partition table begins at sector 2 and ends at sector 33
First usable sector is 34, last usable sector is 4019166
Partitions will be aligned on 1-sector boundaries
Total free space is 0 sectors (0 bytes)

Number  Start (sector)    End (sector)  Size       Code  Name
   1              34              38   2.5 KiB     FFFF  fdt
   2              39             573   267.5 KiB   8300  opensbi
   3             574          169891   82.7 MiB    8300  kernel
   4          169892         4019166   1.8 GiB     8300  filesystem
rudemora@PC-ZBOOK:~/ArTeCS/cvw-main/linux/sdcard$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda      8:0    0 388.4M  1 disk
sdb      8:16   0     2G  0 disk [SWAP]
sdc      8:32   0     1T  0 disk /var/lib/docker
                               /snap
                               /mnt/wslg/distro
                               /
sdd      8:48   1     0B  0 disk
sde      8:64   1   1.9G  0 disk
├─sde1   8:65   1   2.5K  0 part
├─sde2   8:66   1 267.5K  0 part
├─sde3   8:67   1  82.7M  0 part
└─sde4   8:68   1   1.8G  0 part
rudemora@PC-ZBOOK:~/ArTeCS/cvw-main/linux/sdcard$
```

Figure 4.2: File system and SD card partitions.

1. Identify the SD card device (e.g., `/dev/sde`).
2. Launch the partitioning tool:

   ```
   sudo fdisk /dev/sde
   ```

3. Press `n` to create a new partition.
4. Press `Enter` to accept the default partition number.
5. Press `Enter` again to accept the default start sector.
6. Press `Enter` once more to accept the default end sector.
7. Type `w` to write the changes and exit.

After creating the new partition (e.g., `/dev/sde4`), format it using:

```
sudo mkfs.ext4 /dev/sde4
```

Next, mount the partition to a local directory:

```
sudo mkdir -p /mnt/sdcard
sudo mount /dev/sde4 /mnt/sdcard
```

Now copy your desired files or benchmark folders to the mounted directory:

```
sudo cp -r <files> /mnt/sdcard
```

Once the transfer is complete, unmount the partition:

```
sudo umount /mnt/sdcard
```

The files you copied will now be present in the partition on the SD card and accessible from the Linux environment running on the FPGA.

4.1.1.2. Generating the Arty A7 FPGA Bitstream

The process of generating the FPGA bitstream is straightforward thanks to the provided Makefiles and pre-configured build scripts provided in the cvw repository. Vivado, Xilinx's electronic design automation tool, is used to translate the RTL into a hardware configuration bitstream that can be loaded onto the FPGA.

To begin, install Vivado by downloading it from Xilinx's official website at `https://www.xilinx.com/support/download.html`. Note that Vivado is a large toolchain, requiring over 90GiB of disk space.

After installation, you must add support for custom boards by copying the board support files from the repository into the Vivado installation directory. This step is required for proper recognition of the Arty A7 board:

```
sudo cp -r cvw/addins/vivado-boards/new/board_files
   <vivado_path>/<version>/data/boards/
```

The Makefile system supports multiple development boards, with the Arty A7 being the default target. If no target is specified, the build system will generate a bitstream for the Arty A7.

Navigate to the FPGA generator directory:

```
cd cvw/fpga/generator
```

Then, launch the build process by running: `make`

This command will invoke Vivado in batch mode and automatically perform synthesis, implementation, and bitstream generation. During the build, Vivado

may present a command-line prompt. Once the process is complete, you can safely exit the prompt by typing: `exit`

After completion, the resulting `.bit` file and other implementation artifacts will be located in the `fpga/generator` directory. This bitstream can then be used to program the Arty A7 board using Vivado's Hardware Manager.

After completion, the generated `.bit` file and other implementation results will be available under the fpga/generator directory. This bitstream can then be used to program the Arty A7 board using Vivado's Hardware Manager.

4.1.1.3. Programming the Arty A7 FPGA

After generating the bitstream, the next step is to program the Arty A7 FPGA board using Vivado. Begin by launching the Vivado Design Suite and connecting the Arty A7 board to your computer via the UART interface using the J10 port.

To configure the FPGA as the Wally SoC, open Vivado in graphical mode:

```
vivado &
```

In the Vivado GUI, click on "Open Project", then navigate to fpga/generator and select `WallyFPGA.xpr`. Confirm by clicking "OK" (see Figure 4.3).

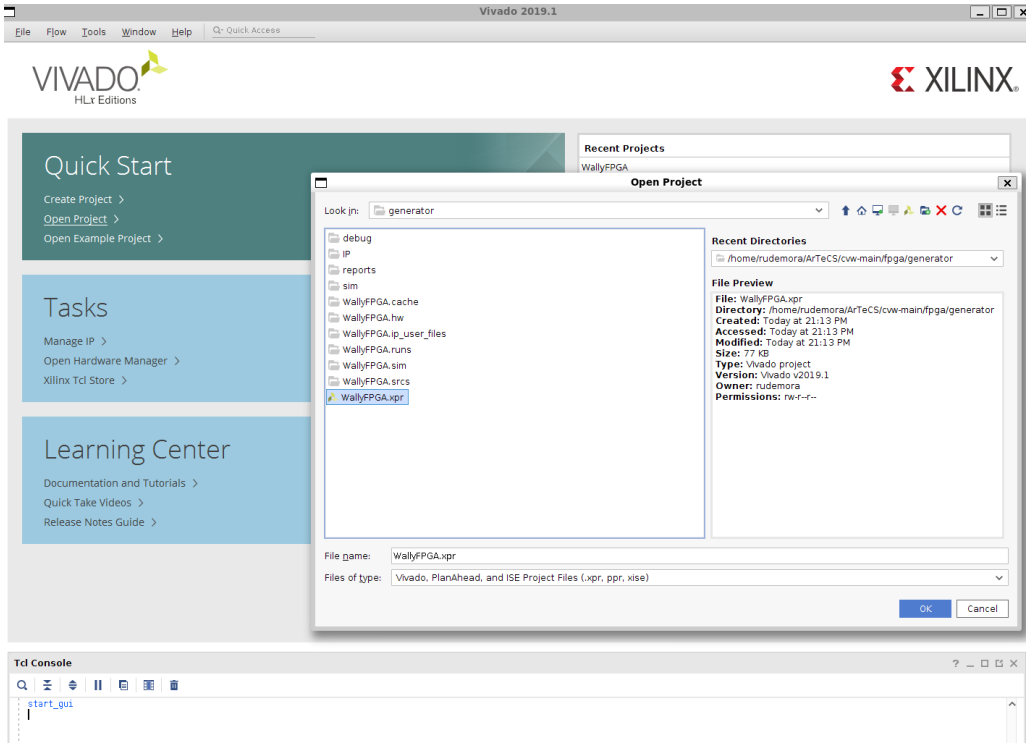


Figure 4.3: Open Wally Project using Vivado.

On the left side of the window, under the "Project Manager" panel, scroll to the bottom and click on "Open Hardware Manager." Before programming the FPGA,

remove the SD card from your computer and insert it into the Digilent PmodSD expansion board. Connect the expansion board to the rightmost Pmod socket on the Arty A7 board (Port D).

Next, connect the USB cable between the FPGA and the host computer. In Vivado, click on "Open Target" followed by "Auto Connect" to establish a connection with the FPGA. The Arty A7 board (with FPGA part number `xc7a35t`) should appear in the device list once the programming interface is detected. Ensure that the board is properly recognized by the system (if using WSL, you may need to make it visible using the same procedure as for the SD card).

Now program the FPGA by selecting "Program Device" and then clicking on the detected FPGA part number in the pop-up dialog. After a few seconds, the FPGA will be programmed with the CVW-Wally core. Once programming completes successfully, the SoC will begin booting the Linux system from the SD card.

4.1.1.4. Connecting a Terminal to the Arty A7 FPGA

After programming the FPGA in Vivado with the Wally bitstream, we can connect a terminal to the FPGA to start using the SoC.

The UART interface on the Arty A7 typically maps to `/dev/ttyUSB1`. To interact with the running system, simply open a serial terminal at a baud rate of `115200`. For example:

```
screen /dev/ttyUSB1 115200
```

If using WSL, the UART device must be made visible in the same way as the SD card reader—using `usbipd` to bind and attach the USB device.

Once the terminal connection is established, the Linux system running on the Wally SoC will begin booting, displaying the system's progress through various initialization stages. These include SD card initialization, partition parsing, and the loading of the device tree, OpenSBI, and the Linux kernel. This process is illustrated in Figure 4.4.

Following the kernel load, control is transferred to OpenSBI, which sets up the runtime environment needed for Linux, as shown in Figure 4.5.

After boot completes, a login prompt will appear, shown in Figure 4.6. The default login is `root`, with no password required. Upon logging in, a minimal Linux environment powered by BusyBox becomes accessible. This shell allows you to execute common Linux commands, check system information, navigate directories, and run benchmark programs or custom scripts from the SD card. As demonstrated in Figure 4.7, the shell behaves like a standard terminal interface, supporting typical commands.

Accessing Files from the SD Card To access the custom partition created earlier and retrieve any files or benchmarks added during preparation, begin by identifying the SD card block device within the Linux environment:

Figure 4.4: Wally booting from SD card: SD card initialization, GPT parsing, and Linux loading.

```
ls /dev/mmcblk*
```

Note: If the partition was previously created as `/dev/sde4` on your host machine, it will likely appear as `/dev/mmcblk0p4` or a similar name once the Linux system re-indexes the partitions internally.

Next, mount the partition and navigate into it:

```
mkdir -p /mnt/sdcard
mount /dev/mmcblk0p4 /mnt/sdcard
cd /mnt/sdcard
```

All previously added files—such as benchmarks, shell scripts, or compiled programs—will now be accessible from the Linux system running on the Wally core. The process is shown in Figure 4.8. After you are done working with the files, unmount the partition to ensure data integrity:

```
umount /mnt/sdcard
```

4.1.2. CVA6 & Genesys 2

To evaluate the CVA6 core on physical hardware, we deployed it on the Digilent Genesys 2 FPGA development board. This implementation follows the official setup provided in the `openhwgroup/cva6` repository, which explicitly supports

Figure 4.5: OpenSBI initializing the platform after boot.

the Genesys 2 as one of its target platforms. The specific version used in this project is `CV32A60X`, as detailed in the official documentation: `https://github.com/openhwgroup/core-v-cores/blob/master/README.md`.

The following equipment is required to complete the deployment process: a Genesys 2 FPGA board, a microSD card and compatible SD card reader, a USB A to Micro-B programming cable, and a host computer with Vivado installed (version 2018.2 is officially supported).

The full deployment process includes four key stages: flashing the Linux image onto the SD card, generating the FPGA bitstream, programming the Genesys 2 board, and finally, establishing a UART terminal connection to interact with the running system.

4.1.2.1. Preparing the SD Card

To boot Linux on the CVA6 system, a compatible Linux image must be built and flashed onto a microSD card. This process is managed through the `cva6-sdk` repos-

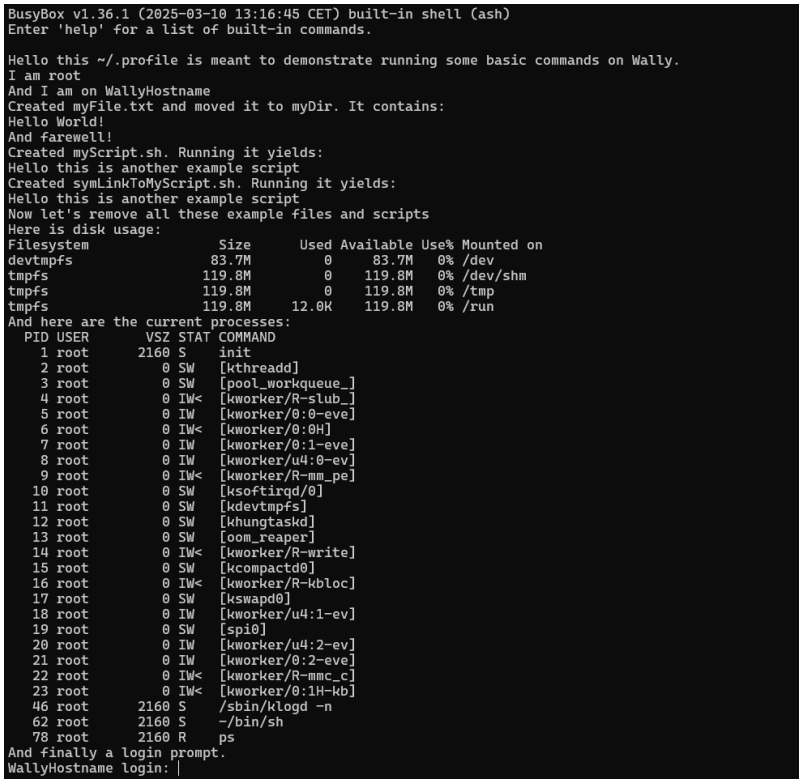


Figure 4.6: Login prompt from BusyBox Linux system running on Wally.

itory, which provides the necessary scripts and tools for generating and installing the image.

Once the SD card reader is accessible (especially if using WSL, as described earlier), follow these steps:

1. Identify the SD device path using `lsblk` or `fdisk -l` (e.g., `/dev/sde`).
2. Clone the cva6-sdk repository:

   ```
   git clone https://github.com/openhwgroup/cva6-sdk
   ```

3. Navigate into the repository and build the Linux image:

   ```
   cd cva6-sdk
   make images
   ```

4. Flash the image to the SD card:

   ```
   sudo -E make flash-sdcard SDDEVICE=/dev/sde
   ```

5. Once complete, safely remove the SD card from the PC.

Figure 4.7: Linux shell accesible after connecting to Wally.

The card is now ready to be inserted into the Genesys 2 FPGA board via the PMOD SD adapter. The system will boot directly from this image once powered on and correctly configured.

4.1.2.2. Generating the Genesys 2 FPGA Bitstream

To deploy the CVA6-based system onto the Genesys 2 FPGA, the hardware design must be synthesized into a format compatible with the board's non-volatile memory. Rather than producing the `.bit` bitstream file for direct programming, the CVA6 flow generates an `.mcs` (Memory Configuration Store) file, which is specifically designed for flashing SPI memory. This format allows the FPGA to boot automatically from the flash memory after power-up, eliminating the need for a

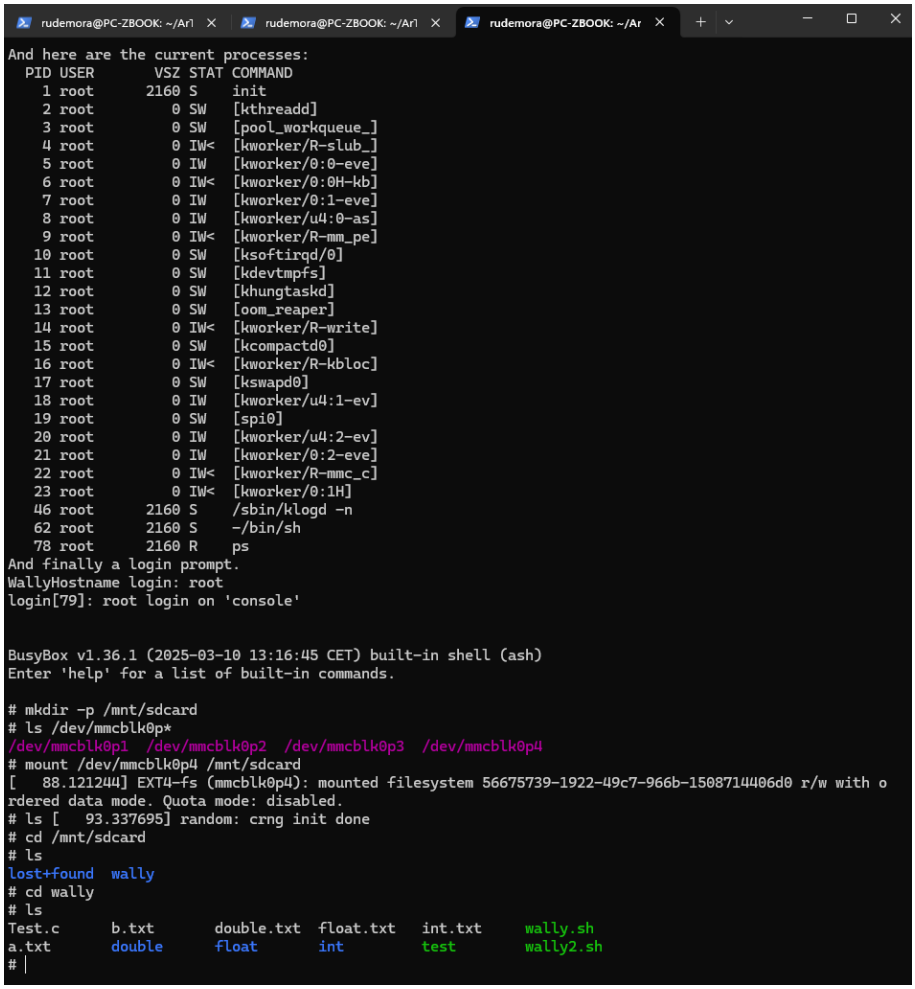


Figure 4.8: SD Card mount in Wally

persistent JTAG connection.

Begin by cloning the official `openhwgroup/cva6` repository. Then, navigate to the following directory:

```
git clone https://github.com/openhwgroup/cva6
cd corev_apu/fpga
```

From this location, run the following command to build the hardware and generate the `.mcs` file. This process can take up to one hour, depending on available resources.

```
make
```

Once the process completes, the resulting `ariane_xilinx.mcs` file will be available for use in the next stage: programming the FPGA.

4.1.2.3. Programming the Genesys 2 FPGA

Launch Vivado (version 2019.1) and open the Hardware Manager. Connect the Genesys 2 board to the host computer using the USB programming cable via JTAG.

To flash the SPI memory, first open the target, then go to `Tools → Add Configuration Memory Device`. Select the Spansion `s25fl256xxxxxx0` flash part. Load the generated `ariane_xilinx.mcs` file, and start the flashing process.

4.1.2.4. Connecting a Terminal to the Genesys 2 FPGA

Once the FPGA has been programmed with the CVA6 bitstream and the SD card is inserted, power on the Genesys 2 board. To interact with the system, connect to the UART interface using a serial terminal such as `screen`:

```
screen /dev/ttyUSB0 115200
```

If you are working within WSL, ensure the UART device is made visible using `usbipd`, following the same procedure described for the SD card reader.

Once the connection is established, the system will begin booting. As shown in Figure 4.9, OpenSBI initializes the runtime environment, providing details about the platform, firmware, and memory configuration. U-Boot then takes over, initializing the peripherals and loading the Linux kernel.

After the kernel boots successfully, the Linux system will continue with hardware initialization and user-space setup, as seen in Figure 4.10. Once the system reaches the login prompt, a Linux environment becomes available. Users can list files, access the filesystem, and execute programs (included in the cva6-sdk repository) such as `tetris`, or other custom applications copied to the SD card.

Focus on Wally Although CVA6 has been used in parallel, our primary calibration target is the Wally core, due to the novelty of its integration and the limited works related to it.

This dual-platform (fpga and gem5) measurement approach enables a meaningful comparison of CPI and instruction behavior, supporting performance tuning and validation of the RISC-V implementation in real hardware and simulation environments.

In the official Wally repository (`cvw-wally`), the `config` directory contains subfolders corresponding to each supported configuration or ISA extension of the Wally core. As mentioned before, for our work, we have focused specifically on the `rv64gc` extension. Within this folder, different Verilog files define the configuration parameters and structural setup of the core. The file `config.vh` specifies the parameters related to that particular extension, such as memory hierarchy, functional unit availability, and pipeline behavior.

Figure 4.9: OpenSBI and U-Boot initializing the CVA6 system on the Genesys 2 FPGA.

After running the synthesis flow using the Makefile provided in the repository and successfully generating the bitstream, a new folder named `deriv` is created. Inside this folder, the subdirectory `fpgaArtyA7` contains another Verilog file, also named `config.vh`. This file holds the final configuration parameters that were used to generate the bitstream for the Arty A7 board. These parameters are essential for achieving an accurate calibration, as they reflect the actual hardware setup used in testing and benchmarking.

4.2. Gem5 Setup

To run architectural simulations and calibrate against FPGA results, we use `gem5`, a powerful and modular simulator widely used in academia and industry. It supports a wide range of ISAs, including RISC-V, and offers both Syscall Emulation (SE) and Full System (FS) simulation modes.

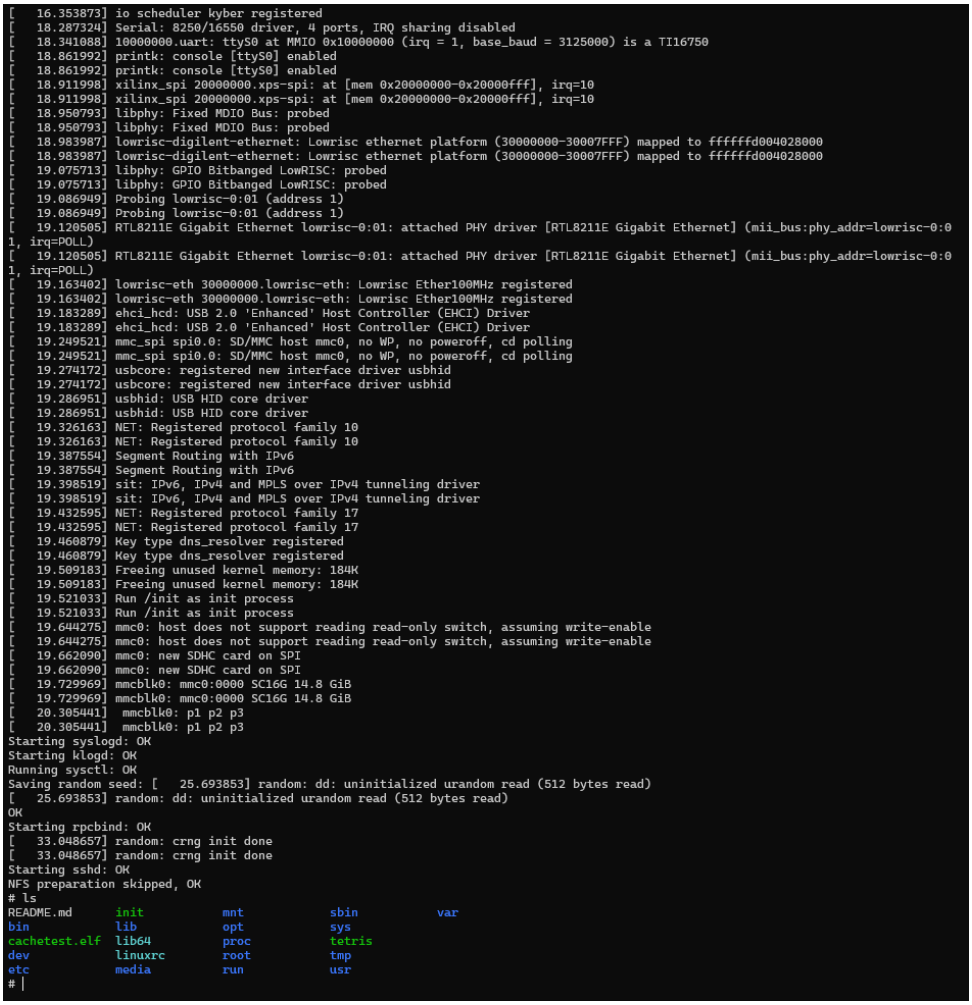


Figure 4.10: Linux boot log on CVA6 showing device initialization and user shell access.

Installing and Building gem5 To set up gem5, the recommended approach is to follow the official tutorial available in the `gem5` repository:

`https://www.gem5.org/documentation/`

The repository provides step-by-step instructions for installing dependencies, building the simulator, and configuring simulation environments.

RISC-V Image Generator (Custom Tool) To simplify the generation of bootable Linux images for FS simulations, we developed a custom RISC-V image builder.

Inspired by previous SD card workflows, this tool automatically builds: a Linux kernel, an OpenSBI bootloader and a root filesystem image

Additionally, any files or executables placed in the internal directories will be accessible once Linux boots inside gem5. This enables quick testing of benchmarks or custom applications.

Simulation Output Once a simulation is complete, gem5 produces a `stats.txt` file that contains detailed statistics, allowing quantitative comparison with real FPGA executions.

4.3. Benchmark: PolyBench

To evaluate the performance of the deployed systems, we used the PolyBench benchmark suite (version 4.2.1), which provides, as described before, a collection of computational kernels organized into different categories.

To compile the suite for the platform, the following steps were carried out:

1. Clone the repository:

   ```
   git clone https://github.com/PolyBench/PolyBenchC-4.2.1
   ```

2. Generate the initial Makefile configuration:

   ```
   cd PolyBenchC-4.2.1
   perl utilities/makefile-gen.pl -cfg
   ```

3. Edit the generated `config.mk` file to configure compilation flags and the toolchain. In our case, the final configuration was:

   ```
   CC=path_to_toolchain
   CFLAGS= -O2 -DPOLYBENCH_USE_C99_PROTO -static -march=rv64gc \
           -DDATA_TYPE_IS_X -DMEDIUM_DATASET
   ```

   We added `-march=rv64gc` to target the appropriate RISC-V ISA extension set and removed the `POLYBENCH_DUMP_ARRAYS` flag to reduce input/output operations. The `DATA_TYPE_IS_X` macro defines the type of data used in the benchmarks, where `X` can be `FLOAT`, `DOUBLE`, or `INT`. If no data type is explicitly specified, the default is `DOUBLE`. In this project, we measured using all three data types to analyze their impact on performance.

   The flag `-static` ensures statically linked binaries, suitable for embedded systems. The `-DMEDIUM_DATASET` macro specifies the input size. We initially experimented with `MINI` and `SMALL` datasets for fast verification and debugging. However, we ultimately selected `MEDIUM_DATASET` as the most suitable size for our evaluation: it increased simulation time enough to make performance differences measurable, but did not cause excessive slowdown or execution overhead.

   All benchmarks were compiled with `-O2`, a standard optimization level that balances compile time and runtime efficiency. This optimization setting is particularly appropriate for performance analysis, as it produces realistic execution behavior without enabling aggressive transformations that might overly distort control flow or memory access patterns.

4. Compile all benchmarks:

   ```
   perl utilities/run-all.pl .
   ```

   This script builds all benchmarks automatically. While it may produce execution errors on x86 hosts (due to incompatible RISC-V binaries), this is expected and does not affect functionality for the target platform.

4.4. Calibration process

To evaluate and calibrate the performance of the Wally core implemented on the Arty A7 FPGA, we compare its behavior against simulations executed in `gem5`. The primary objective is to validate the simulation model by aligning it with the behavior of the real hardware, which serves as the reference baseline.

FPGA Performance Measurement In the FPGA, performance counters are enabled for the Wally core. These include:

- `cycle`: Counts the number of executed clock cycles.
- `instret`: Counts the number of retired instructions.

These two counters were selected because they are accessible from user mode, which simplifies measurement within the Linux environment. However, many more performance counters are available in machine mode, offering deeper microarchitectural insight when privileged access is possible. A broader set of machine-level performance counters and their integration with standard Linux profiling tools such as `perf` are discussed in the work Domingos et al. (2021).

We use a custom C program that reads these hardware counters before and after executing a benchmark. The values are read using inline assembly and written to a results file. This process is repeated multiple times for each benchmark to obtain reliable average metrics.

The counters are accessed through standard RISC-V CSRs (`cycle`, `instret`). The program must be statically compiled using a RISC-V toolchain. This allows us to run it directly on the RISC-V Linux system running on the FPGA.

As part of the calibration process for Wally, we tested the three available data types in PolyBench: double, float, and int. To do this, we compiled the PolyBench suite for each data type and copied the resulting binaries to the SD card used by the FPGA. A directory named Wally was created on the SD card, containing three subdirectories (one for each data type) along with a custom C program and its compiled binary. This C program is responsible for measuring performance counters before and after benchmark execution.

To automate the calibration process, we also included a Bash script that sequentially runs the same benchmark 10 times for each data type. An example of this script is shown in the Figure 4.12, and the C program is shown in the Figure 4.11.

For example, running the command in the Linux prompt inside Wally:

```
sh wally.sh inMedium/linear-algebra/blas/gemm/gemm outMedium/gemm.txt
```

will execute the benchmark located at the specified path 10 times for each data type. The bash script simply automates the execution flow, while the custom C program captures the performance counters and appends the results cycles, instruction count, and to output files. Figure 4.13 shows an example of execution.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
   long long int cycle_val1;
   long long int cycle_val2;
   long long int instr1;
   long long int instr2;
   char *input = argv[1];
   asm volatile ("csrr %0, cycle" : "=r"(cycle_val1));
   asm volatile ("csrr %0, instret" : "=r"(instr1));
   system(input);
   asm volatile ("csrr %0, cycle" : "=r"(cycle_val2));
   asm volatile ("csrr %0, instret" : "=r"(instr2));
   char *output = argv[2];
   FILE *file = fopen(output, "a");
   fprintf(file, "--------------------------\n");
   fprintf(file, "Cycles: %lld\n", cycle_val2 - cycle_val1);
   fprintf(file, "Instr number: %lld\n", instr2 - instr1);
   fprintf(file, "CPI: %.3f\n", (double)(cycle_val2 - cycle_val1) / (instr2 - instr1));
   fclose(file);
}
```

Figure 4.11: Custom C program used to collect cycle and instruction counts

Gem5 Performance Measurement In gem5, we use the Full System (FS) mode with Linux and the same compiled benchmarks. The measurement procedure consists of:

- Issuing `m5 resetstats` before benchmark execution.
- Running the benchmark.
- Issuing `m5 dumpresetstats` after execution.

This captures the simulation statistics into the `m5out/stats.txt` file. While gem5 outputs a vast set of statistics, for calibration purposes we focus only on:

```
#!/bin/bash
INPUT="$1"
OUTPUT="$2"
for i in $(seq 1 10); do
  echo "Simulation $i:"
  ./test ./double/"$INPUT" ./double/"$OUTPUT"
  ./test ./float/"$INPUT" ./float/"$OUTPUT"
  ./test ./int/"$INPUT" ./int/"$OUTPUT"
  echo "Simultion $i finished."
  echo "----------------------------"
done

echo "Execution finished"
```

Figure 4.12: Bash script to automate benchmark execution for calibration

- `simInsts`: Number of simulated instructions.
- `system.cpu.numCycles`: Number of simulated clock cycles.

As with the FPGA setup, each benchmark is executed multiple times to collect consistent performance data. However, in this case, a custom C program is not required to access hardware counters. Instead, gem5 provides detailed performance statistics in the automatically generated m5out/stats.txt file.

The calibration process can be automated using a Bash script that begins by resetting the statistics, executes the benchmark, and then dumps the results—repeating this process for each run.

These commands allow us to isolate the measures, discarding initialization overhead (resetstats) and focusing only on the actual benchmark execution. Figures 4.14 and 4.15 show an example of the beginning and end of a statistics section generated by these commands.

CPU Model In gem5, there are several CPU models available, each offering different levels of accuracy and complexity. These include `AtomicSimpleCPU` and `TimingSimpleCPU`, which are mainly used for fast functional simulation; `O3CPU`, a highly accurate out-of-order model; and `MinorCPU`, with a fixed pipeline but configurable data structures and execution behavior.

Since Wally is an in-order processor, we chose to base our gem5 simulation model on `MinorCPU`. We created a new Python class called `WallyCPU`, extending `MinorCPU`,

to serve as our base model for calibration. Within the `WallyCPU` class, we not only instantiate the processor model but also override several internal parameters to align the behavior of the simulator with the characteristics of the real hardware. The initial simulation system used for this calibration work is a full-system gem5 setup, shown in Figure 4.16, which operates without caches and uses gem5's default parameter values.

As Wally supports the `rv64gc` ISA extension set, we verified that gem5 uses the same configuration by inspecting the output of `/proc/cpuinfo` within the simulated system (Figure 4.17). This ensures that both environments operate under the same architectural assumptions. From this starting point, a series of iterative adjustments were made to progressively align the simulator's behavior with that of the actual FPGA implementation. The main calibration parameters modified include:

- Cache configuration: a single level of caches was integrated in the system. Both instruction and data caches were configured with 16KB capacity, 4-way associativity, and 64-byte cache line size. Cache access latency was reduced from 2 to 1 cycle as that's the value in Wally.
- TLBs: The default gem5 configuration uses TLBs of size 64 entries. For both instruction and data TLBs, we reduced this to 32 entries to match the hardware configuration.
- Branch prediction: Wally employs a gshare branch predictor but gem5 doesn't include the implementation by default. We integrated an external implementation[2]. In our configuration, the branch target buffer (BTB) consists of 1024 entries, with 10 bits used for the tag and a total of 16 entries in the predictor table.
- Pipeline widths: Since Wally issues, executes, and commits a single instruction per cycle, we set decodeInputWidth = 1, executeInputWidth = 1, executeIssueLimit = 1, executeCommitLimit = 1.
- Main memory latency: To replicate the memory hierarchy more accurately, the latency for main memory accesses was adjusted to 20 ns, equivalent to 40 cycles at the 2 GHz clock frequency shared by both the gem5 model and the board.

  To estimate the latency of main memory, we designed a simple assembly-level microbenchmark that forces repeated memory accesses likely to miss in cache, thereby incurring the full latency of main memory. The test consists of the two small routines `Init_page` and `Cache_test`, shown below.

  Listing 4.1: Memory access microbenchmark in RISC-V assembly

```
.globl Cache_test
.globl Init_page
.section .data
```

[2] `https://github.com/karnapathak/gem5/tree/3aa48f51403a0d18743a316f2b8bd3919e76d766/src/cpu/pred`

```
  .align 12
data_array:
    .space 1024

.section .text

Cache_test:
    la t0, data_array
    li t1, 0
    li t2, 896
    li t3, 64
loop:
    lw t5, 64(t0)
    add t0, t0, t3
    add t1, t1, t3
    blt t1, t2, loop
    ret

Init_page:
    la t0, data_array
    lw t5, 0(t0)
    lw t5, 1020(t0)
    ret
```

The `Init_page` function ensures that the pages containing `data_array` are mapped and ready, avoiding page faults or TLB misses during execution. It accesses both the first and last words of the array to guarantee that relevant memory pages (the size of the pages is 16KB) are brought into memory. The `Cache_test` function performs 14 iterations of memory accesses, each spaced by 64 bytes, forcing the processor to fetch data from main memory.

As we did before for executing benchmarks, we created a simple program in C, (shown in Figure 4.18) that uses inline assembly to access the RISC-V performance counters `cycle` and `instret`. These counters allow us to record the number of executed cycles and retired instructions before and after executing the assembly subroutine.

This program enables automated measurement of cycles, instruction count, and CPI for any executable. It is compiled statically using the RISC-V toolchain to ensure the compatibility with the embedded Linux system running on the FPGA.

- Total number of cycles: 650
- Total number of instructions: 67
- Number of iterations: 14

Note: The number of cycles and instructions were measured on the FPGA using the performance counters as described before.

Cycle Breakdown: From Verilator simulation traces (Figure 4.19) and instruction disassembly (Figure 4.20), we measured:

Each iteration involves approximately 6 cycles of fixed overhead:

- 4 cycles from instruction execution (`lw`, two `add`s, `blt`)
- 2 cycles from branch misprediction

Thus, we model the total cycle count as:

$$Totalcycles = 14 \times (6 + L_{mem}) = 650$$

This yields an estimated main memory latency of **40 cycles**, which corresponds to a 20 ns latency at a 2 GHz clock frequency.

- Functional units: the latencies for functional units were obtained through cycle-level execution traces using Verilator. By analyzing the simulation output and observing the number of cycles taken by specific operations, we determined realistic latency values.

  - Integer multiplication: 2 cycles
  - Integer division: 5 cycles
  - Floating-point operations: 2 cycles. For floating-point divison and square root, we left the default value of 6 cycles.

As an illustrative example, we now examine how the 2-cycle latency for integer multiplication was determined.

Figure 4.21 shows the execution of a simple loop containing both arithmetic and multiplication instructions. The corresponding disassembly is listed below:

```
100c4: 00138393   addi t2, t2, 1
100c8: 03de0f33   mul  t5, t3, t4
100cc: 001f0593   addi a1, t5, 1
100d0: 00158613   addi a2, a1, 1
100d4: ff03c8e3 blt t2,a6,100c4 <REPEAT>
```

The signals captured in the waveform include:

- `InstrD[31:0]`: Instruction in Decode stage
- `A, B, ALUResultE`: Operands and ALU output in Execute stage
- `ProdM`: Output of the multiplier in the Memory stage
- `ForwardedSrcBE`, `ForwardedSrcAE`: Values forwarded for data dependencies

The key observation comes from analyzing the data dependency (t5 register) between the multiplication and addition instruction.

Here, the `addi` instruction depends on the result of the previous `mul`, which writes to `t5`. Although `addi` is an integer operation with a 1-cycle latency, it stalls for two cycles in the Decode (D) stage. This stall occurs because the operand `t5` is not yet available. It is still being produced by the multiplication unit.

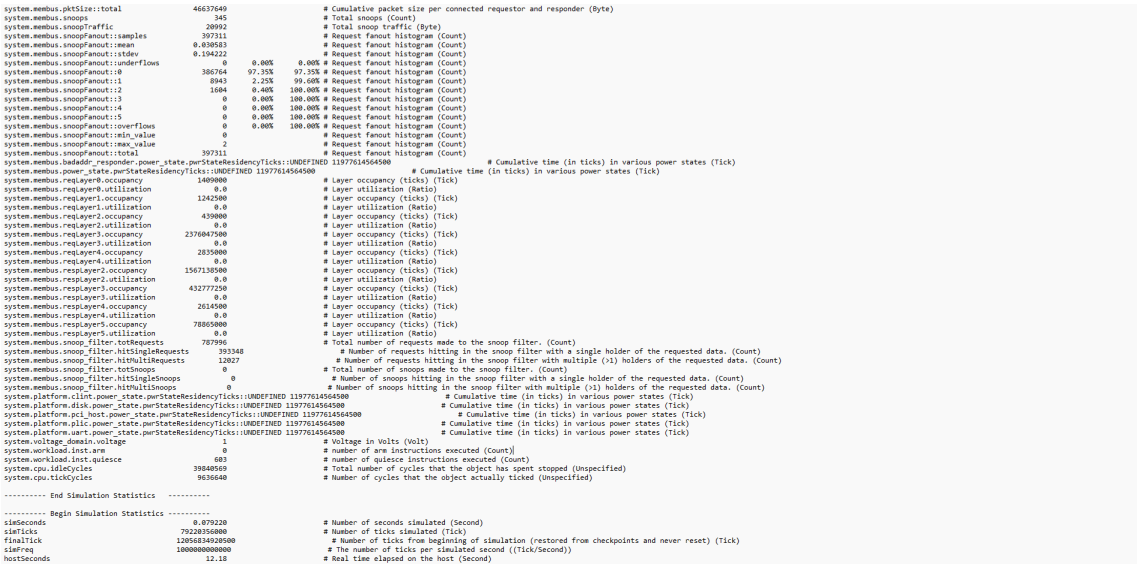
The fact that `addi` waits two cycles in Decode before proceeding to Execution confirms that the multiplication operation requires two cycles to produce its result.

```
# rm float/outMedium/gemm.txt
# cat wally.sh
#!/bin/bash

if [ $# -ne 2 ]; then
  echo "Uso: $0 <nombre> <parámetro>"
  exit 1
fi

NOMBRE="$1"
NOMBRE2="$2"
for i in $(seq 1 5); do
  echo "Simulación $i:"
  echo "float"
  ./test ./float/"$NOMBRE" ./float/"$NOMBRE2"
  #echo "int"
  ./test ./int/"$NOMBRE" ./int/"$NOMBRE2"
  echo "Simulación $i terminada."
  echo "------------------------------"
done

echo "Todas las simulaciones han finalizado."
# sh wally.sh inMedium/linear-algebra/blas/gemm/gemm outMedium/gemm.txt
Simulación 1:
float
Simulación 1 terminada.
------------------------------
Simulación 2:
float
Simulación 2 terminada.
------------------------------
Simulación 3:
float
Simulación 3 terminada.
------------------------------
Simulación 4:
float
Simulación 4 terminada.
------------------------------
Simulación 5:
float
Simulación 5 terminada.
------------------------------
Todas las simulaciones han finalizado.
# cat int/outMedium/gemm.txt
---------------------------
Ciclos: 1787551696
Num instr: 636437588
CPI: 2.809
---------------------------
Ciclos: 174393102
Num instr: 114602687
CPI: 1.522
---------------------------
Ciclos: 211762967
Num instr: 127198799
CPI: 1.665
---------------------------
Ciclos: 173822205
Num instr: 114483649
CPI: 1.518
---------------------------
Ciclos: 191670552
Num instr: 120512883
CPI: 1.590
#
```

Figure 4.13: Wally execution of Gemm benchmark

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.248682                       # Number of seconds simulated (Second)
simTicks                                   248681991000                     # Number of ticks simulated (Tick)
finalTick                                 1197614564500                     # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                   1000000000000                     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                                 20.79                           # Real time elapsed on the host (Second)
hostTickRate                              11962995012                       # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                                   839104                         # Number of bytes of host memory used (Byte)
simInsts                                    6249150                         # Number of instructions simulated (Count)
simOps                                      6288125                         # Number of ops (including micro ops) simulated (Count)
hostInstRate                                 300618                         # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                                   302493                         # Simulator op (including micro ops) rate (op/s) ((Count/Second))
system.bridge.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.clk_domain.clock                      1000                           # Clock period in ticks (Tick)
system.cpu.numCycles                        49477209                        # Number of cpu cycles simulated (Cycle)
system.cpu.cpi                              7.917430                        # CPI: cycles per instruction (core level) ((Cycle/Count))
system.cpu.ipc                              0.126304                        # IPC: instructions per cycle (core level) ((Count/Cycle))
system.cpu.numWorkItemsStarted                   0                          # Number of work items this cpu started (Count)
system.cpu.numWorkItemsCompleted                 0                          # Number of work items this cpu completed (Count)
system.cpu.quiesceCycles                    447886773                       # Total number of cycles that CPU has spent quiesced or waiting for an interrupt (Cycle)
system.cpu.branchPred.lookups_0::NoBranch         1071      0.07%     0.07% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::Return         191637     12.79%    12.86% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::CallDirect      129780      8.66%    21.52% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::CallIndirect       64470      4.30%    25.83% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::DirectCond       956152     63.81%    89.64% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::DirectUncond       131533      8.78%    98.42% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::IndirectCond            0      0.00%    98.42% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::IndirectUncond        23721      1.58%   100.00% # Number of BP lookups (Count)
system.cpu.branchPred.lookups_0::total        1498364                       # Number of BP lookups (Count)
system.cpu.branchPred.squashes_0::NoBranch          113      0.03%      0.03% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::Return        30132      9.31%      9.34% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::CallDirect        33500     10.35%     19.69% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::CallIndirect        18943      5.85%     25.54% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::DirectCond       199073     61.49%     87.03% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::DirectUncond        38836     12.00%     99.03% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::IndirectCond            0      0.00%     99.03% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::IndirectUncond         3146      0.97%    100.00% # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.squashes_0::total       323743                        # Number of branches that got squashed (completely removed) as an earlier branch was mispredicted. (Count)
system.cpu.branchPred.corrected_0::NoBranch          957      0.68%      0.68% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::Return        39549     24.99%     25.68% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a corrected
branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::CallDirect        18657     11.79%     37.39% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::CallIndirect        10433      6.59%     43.98% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::DirectCond        71674     45.30%     89.28% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::DirectUncond        15539      9.82%     99.10% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::IndirectCond            0      0.00%     99.10% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::IndirectUncond         1422      0.90%    100.00% # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a
corrected branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.corrected_0::total       158231                        # Number of branches that got corrected but not yet commited. Branches get corrected by decode or after execute. Also a branch misprediction can be detected out-of-order. Therefore, a corrected
branch might not end up beeing committed in case an even earlier branch was mispredicted (Count)
system.cpu.branchPred.earlyResteers_0::NoBranch            0                # Number of branches that got redirected after decode. (Count)
system.cpu.branchPred.earlyResteers_0::Return            0                  # Number of branches that got redirected after decode. (Count)
system.cpu.branchPred.earlyResteers_0::CallDirect            0              # Number of branches that got redirected after decode. (Count)
system.cpu.branchPred.earlyResteers_0::CallIndirect            0            # Number of branches that got redirected after decode. (Count)
```

Figure 4.14: Beginning of stats section in Gem5



```
system.membus.pktSize::total                 46637649                       # Cumulative packet size per connected requestor and responder (Byte)
system.membus.snoops                           345                          # Total snoops (Count)
system.membus.snoopTraffic                   20992                          # Total snoop traffic (Byte)
system.membus.snoopFanout::samples           397311                         # Request fanout histogram (Count)
system.membus.snoopFanout::mean            0.030583                         # Request fanout histogram (Count)
system.membus.snoopFanout::stdev           0.194222                         # Request fanout histogram (Count)
system.membus.snoopFanout::underflows            0      0.00%      0.00% # Request fanout histogram (Count)
system.membus.snoopFanout::0                385764     97.35%     97.35% # Request fanout histogram (Count)
system.membus.snoopFanout::1                  8943      2.25%     99.60% # Request fanout histogram (Count)
system.membus.snoopFanout::2                  1604      0.40%    100.00% # Request fanout histogram (Count)
system.membus.snoopFanout::3                     0      0.00%    100.00% # Request fanout histogram (Count)
system.membus.snoopFanout::4                     0      0.00%    100.00% # Request fanout histogram (Count)
system.membus.snoopFanout::5                     0      0.00%    100.00% # Request fanout histogram (Count)
system.membus.snoopFanout::overflows             0      0.00%    100.00% # Request fanout histogram (Count)
system.membus.snoopFanout::min_value             0                       # Request fanout histogram (Count)
system.membus.snoopFanout::max_value             2                       # Request fanout histogram (Count)
system.membus.snoopFanout::total            397311                       # Request fanout histogram (Count)
system.membus.badaddr_responder.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.membus.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.membus.reqLayer0.occupancy           1409000                          # Layer occupancy (ticks) (Tick)
system.membus.reqLayer0.utilization             0.0                          # Layer utilization (Ratio)
system.membus.reqLayer1.occupancy           1242500                          # Layer occupancy (ticks) (Tick)
system.membus.reqLayer1.utilization             0.0                          # Layer utilization (Ratio)
system.membus.reqLayer2.occupancy            439000                          # Layer occupancy (ticks) (Tick)
system.membus.reqLayer2.utilization             0.0                          # Layer utilization (Ratio)
system.membus.reqLayer3.occupancy        2376047500                          # Layer occupancy (ticks) (Tick)
system.membus.reqLayer3.utilization             0.0                          # Layer utilization (Ratio)
system.membus.reqLayer4.occupancy           2835000                          # Layer occupancy (ticks) (Tick)
system.membus.reqLayer4.utilization             0.0                          # Layer utilization (Ratio)
system.membus.respLayer2.occupancy        1567130500                         # Layer occupancy (ticks) (Tick)
system.membus.respLayer2.utilization            0.0                          # Layer utilization (Ratio)
system.membus.respLayer3.occupancy         432777250                         # Layer occupancy (ticks) (Tick)
system.membus.respLayer3.utilization            0.0                          # Layer utilization (Ratio)
system.membus.respLayer4.occupancy          2614500                          # Layer occupancy (ticks) (Tick)
system.membus.respLayer4.utilization            0.0                          # Layer utilization (Ratio)
system.membus.respLayer5.occupancy         78865000                          # Layer occupancy (ticks) (Tick)
system.membus.respLayer5.utilization            0.0                          # Layer utilization (Ratio)
system.membus.snoop_filter.totRequests         787996                        # Total number of requests made to the snoop filter. (Count)
system.membus.snoop_filter.hitSingleRequests       393348                    # Number of requests hitting in the snoop filter with a single holder of the requested data. (Count)
system.membus.snoop_filter.hitMultiRequests        12027                     # Number of requests hitting in the snoop filter with multiple (>1) holders of the requested data. (Count)
system.membus.snoop_filter.totSnoops               0                         # Total number of snoops made to the snoop filter. (Count)
system.membus.snoop_filter.hitSingleSnoops          0                        # Number of snoops hitting in the snoop filter with a single holder of the requested data. (Count)
system.membus.snoop_filter.hitMultiSnoops           0                        # Number of snoops hitting in the snoop filter with multiple (>1) holders of the requested data. (Count)
system.platform.clint.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.platform.disk.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.platform.pci_host.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.platform.plic.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.platform.uart.power_state.pwrStateResidencyTicks::UNDEFINED 1197614564500   # Cumulative time (in ticks) in various power states (Tick)
system.voltage_domain.voltage                   1                            # Voltage in Volts (Volt)
system.workload.inst.arm                        0                            # number of arm instructions executed (Count)
system.workload.inst.quiesce                  683                            # number of quiesce instructions executed (Count)
system.cpu.idleCycles                       39840569                         # Total number of cycles that the object has spent stopped (Unspecified)
system.cpu.tickCycles                        9636640                         # Number of cycles that the object actually ticked (Unspecified)

---------- End Simulation Statistics   ----------

---------- Begin Simulation Statistics ----------
simSeconds                                   0.079220                       # Number of seconds simulated (Second)
simTicks                                   79220356000                      # Number of ticks simulated (Tick)
finalTick                                1205683492500                      # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                   1000000000000                     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                                 12.18                           # Real time elapsed on the host (Second)
```

Figure 4.15: End of stats section in Gem5

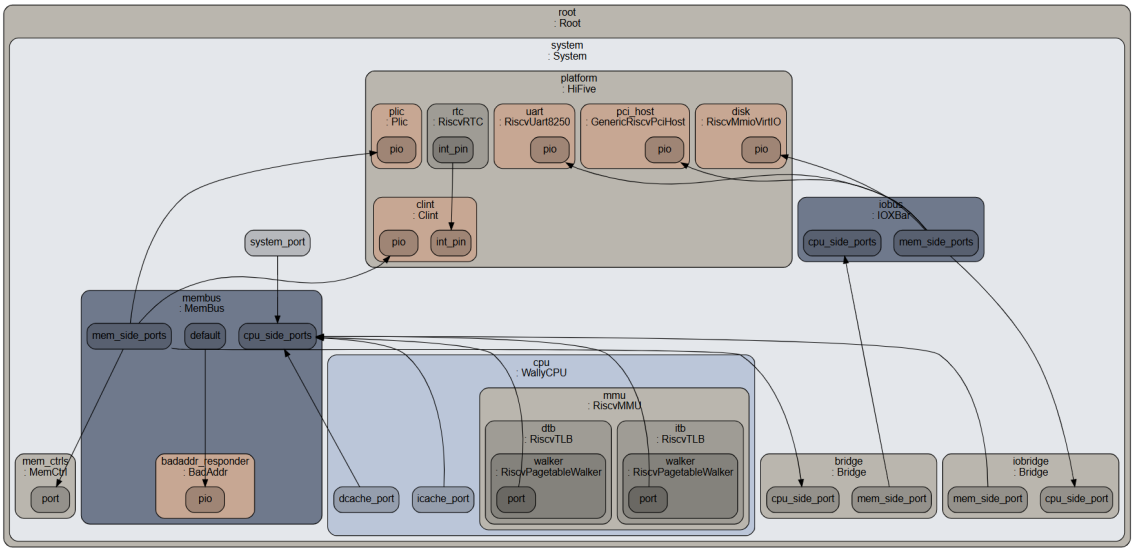


Figure 4.16: Initial Gem5 System.

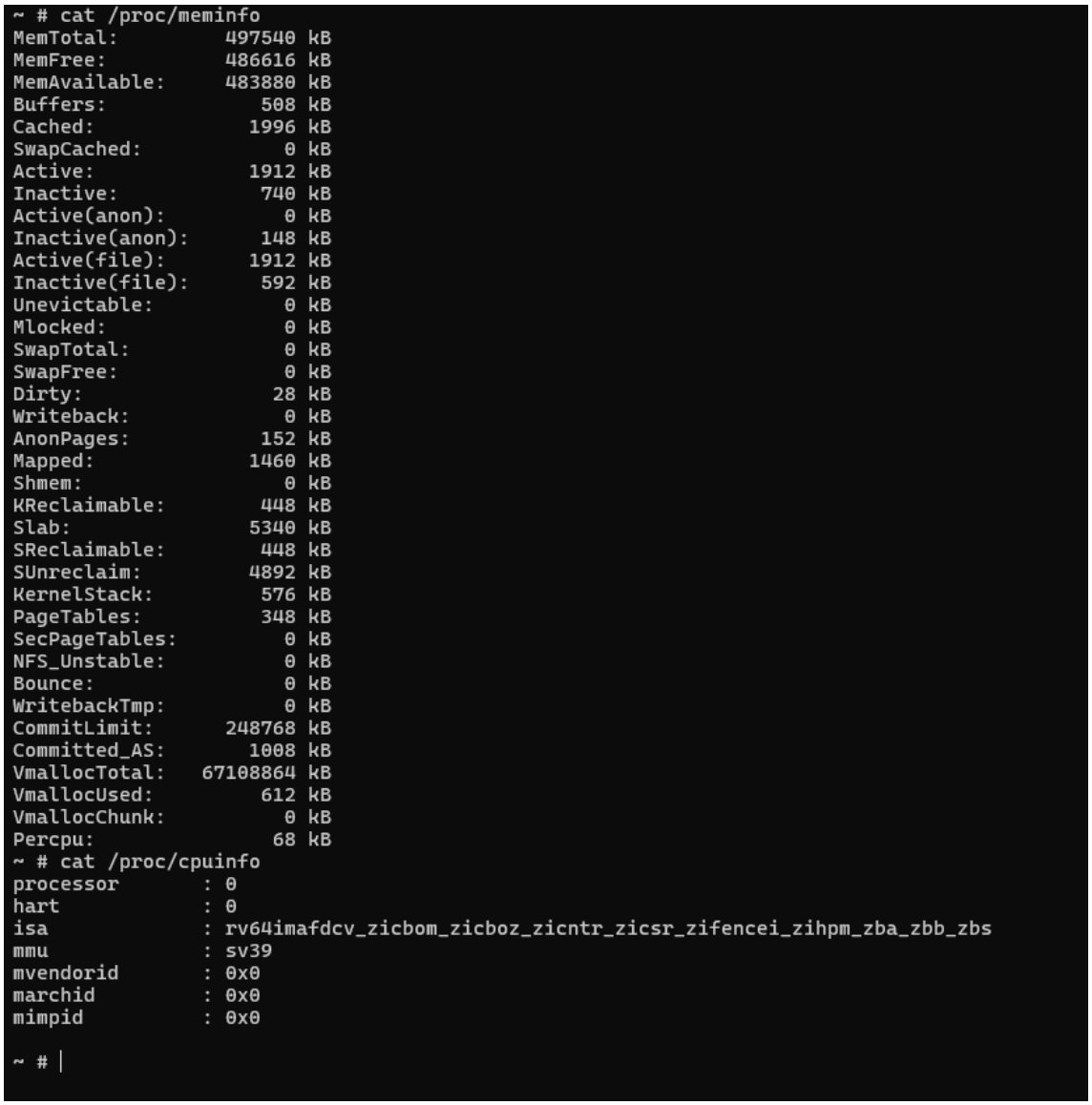


Figure 4.17: Gem5 extension supported.

```c
#include <stdio.h>
#include <stdlib.h>

extern void Cache_test(void);
extern void Init_page(void);

int main(void)
{
    Init_page();
    long long int cycle_val1;
    long long int cycle_val2;
    long long int instr1;
    long long int instr2;
    asm volatile ("csrr %0, cycle" : "=r"(cycle_val1));
    asm volatile ("csrr %0, instret" : "=r"(instr1));
    Cache_test();
    asm volatile ("csrr %0, cycle" : "=r"(cycle_val2));
    asm volatile ("csrr %0, instret" : "=r"(instr2));
    printf("Total cycles: %lld\n", cycle_val2 - cycle_val1);
    printf("Total instr: %lld\n", instr2 - instr1);
    printf("CPI: %.3f\n", (double)(cycle_val2 - cycle_val1) / (instr2 - instr1));
    return 0;
}
```

Figure 4.18: C program for collecting cycle and instruction counts using RISC-V performance counters.

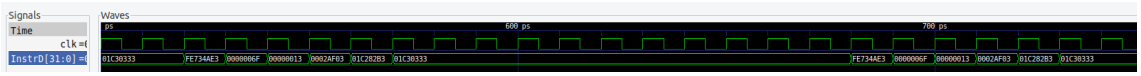


Figure 4.19: Verilator waveform.

```
Disassembly of section .text:

00000000000100e8 <rvtest_entry_point>:
   100e8:   02a00513              li a0,42
   100ec:   00001297              auipc t0,0x1
   100f0:   03428293              addi  t0,t0,52 # 11120 <__DATA_BEGIN__>
   100f4:   00000313              li t1,0
   100f8:   000013b7              lui   t2,0x1
   100fc:   04000e13              li t3,64

0000000000010100 <loop>:
   10100:   0002af03              lw t5,0(t0)
   10104:   01c282b3              add   t0,t0,t3
   10108:   01c30333              add   t1,t1,t3
   1010c:   fe734ae3              blt   t1,t2,10100 <loop>

0000000000010110 <self_loop>:
   10110:   0000006f              j  10110 <self_loop>
```

Figure 4.20: Disassembly of the benchmark program used for memory latency measurement.

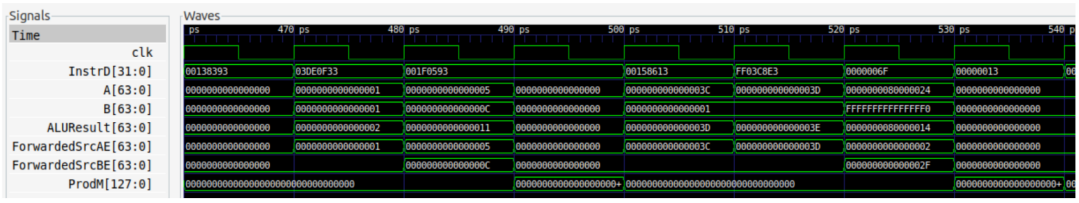


Figure 4.21: Waveform showing execution of a `mul` followed by dependent `addi`. The multiplication result appears in the Memory stage, causing a pipeline bubble.

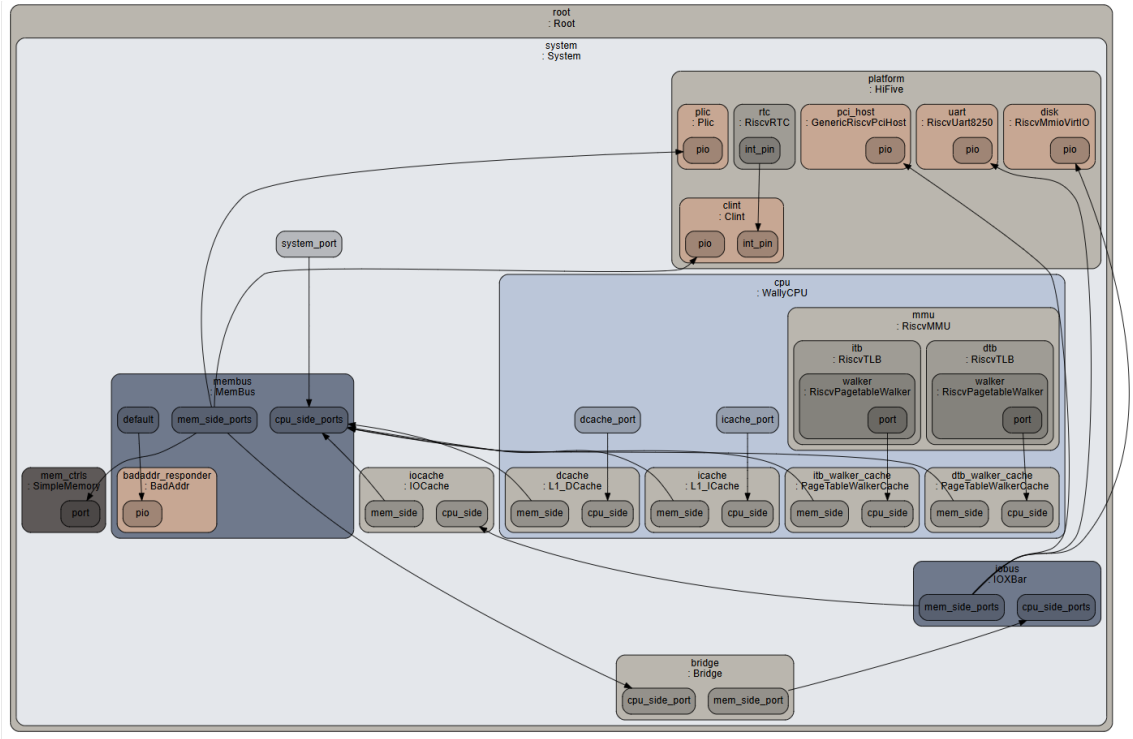


Figure 4.22: Final Gem5 System.

Chapter 5

Results

To validate and calibrate the simulated behavior of the Wally core in `gem5` against its FPGA implementation, we conducted extensive benchmark experiments across a variety of architectural configurations. Each configuration was evaluated using the medium dataset size from PolyBench, compiled statically with the `rv64gc` ISA extension and integer data type. For each run, the number of executed instructions, total clock cycles, and resulting CPI (cycles per instruction) were measured.

5.1. Gemm

To guide the simulation calibration process, we focused on the gemm benchmark from PolyBench, which performs matrix multiplication of the form $\alpha AB + \beta C$ using the default double data type. This kernel is computationally intensive and memory-access heavy, making it ideal for validating both functional and performance behavior.

The benchmark was executed directly on the Wally core running on the Arty A7 FPGA. As shown in Figure 5.1, we ran the benchmark 20 times and discarded the first execution, which consistently exhibited irregular performance due to initialization effects. From the remaining iterations, we observed very stable behavior. The average performance was approximately:

- 120 million instructions
- 225 million cycles
- CPI ≈ 1.88

Initial Simulation State We began the simulation from an untuned configuration with default `gem5` parameters. The average of 20 independent executions was:

```
simInsts 116,590,404  numCycles 3,484,514,438  CPI = 29.91
```

```
# time ./test inMedium/linear-algebra/blas/gemm/gemm outMedium/gemm.txt
real    2m 18.32s
user    0m 10.81s
sys     0m 58.01s
# time ./test inMedium/linear-algebra/blas/gemm/gemm outMedium/gemm.txt
real    0m 10.71s
user    0m 8.24s
sys     0m 0.81s
# time ./test inMedium/linear-algebra/blas/gemm/gemm outMedium/gemm.txt
real    0m 11.69s
user    0m 8.27s
sys     0m 0.89s
# time ./test inMedium/linear-algebra/blas/gemm/gemm outMedium/gemm.txt
real    0m 11.15s
user    0m 8.28s
sys     0m 0.73s
# cat outMedium/gemm.txt
--------------------------
Ciclos: 1571658346
Num instr: 557422304
CPI: 2.820
--------------------------
Ciclos: 212633358
Num instr: 117522431
CPI: 1.809
--------------------------
Ciclos: 232366932
Num instr: 124076775
CPI: 1.873
--------------------------
Ciclos: 221436229
Num instr: 120355497
CPI: 1.840
#
```

Figure 5.1: Execution of `gemm` benchmark on Wally with double data type. First run is discarded.

The simulation CPI was more than an order of magnitude greater than the FPGA reference, highlighting the need for progressive model refinement.

Progressive Model Tuning

- Adding Cache Support. Separate instruction and data L1 caches were integrated in the simulation model using gem5's default configuration. Even without tuning, this addition had a major impact, significantly reducing the CPI:

  `simInsts 182,250,929  numCycles 715,411,680  CPI = 3.93`

- The next step involved configuring the caches to match the specification provided in the Wally documentation. After parametrizing the caches with 16KB size, 4-way associativity, and 1-cycle latency:

  `simInsts 109,498,803  numCycles 391,958,836  CPI = 3.58`

- Modeling TLBs. Once instruction and data TLBs were limited to 32 entries, we observed:

  ```
  simInsts 104,156,874  numCycles 341,307,490  CPI = 3.28
  ```

- A GShare predictor was integrated using parameters similar to Wally's: 1024 entries, 10-bit tags, and 16 entries.

  ```
  simInsts 102,663,086  numCycles 325,352,648  CPI = 3.17
  ```

- Functional unit latencies were adjusted to match Verilator-measured values, maintaining 2 integer ALUs and allowing up to 2 instructions per decode, issue, and commit:

  ```
  simInsts 104,382,281  numCycles 327,045,976  CPI = 3.13
  ```

- Reduced Pipeline Width: limiting pipeline bandwidth to 1 instruction per decode/execute/commit stage and reducing integer ALUs to 1.

  ```
  simInsts 109,593,925  numCycles 284,789,512  CPI = 2.60
  ```

- Finally, after tuning main memory latency to 20ns (corresponding to 40 cycles at 2GHz):

  ```
  simInsts 100,309,639  numCycles 229,479,512  CPI = 2.29
  ```

Conclusion After iterative tuning and architectural constraint enforcement, the simulation CPI was reduced from 28.6 to 2.29, closely aligning with the measured FPGA average of 1.82. The remaining gap is attributed to factors such as modeling limitations in `gem5`, I/O operations, and parameters not calibrated.

As can be seen from Figures 5.2, 5.3, and 5.4, the calibration process yields a simulation model that not only captures global CPI trends but also approximates instruction and cycle counts with strong fidelity. The consistency across all three data types, `int`, `float`, and `double`, demonstrates that the calibrated configuration perf single use case, providing a reliable foundation for further experimentation and architectural exploration.

This benchmark demonstrates the effectiveness of the full calibration process. From the initial configuration with extremely high CPI values to the final setup, the improvement is substantial. For the `double` data type, the CPI deviation was approximately 19.8%, showing particularly strong alignment.

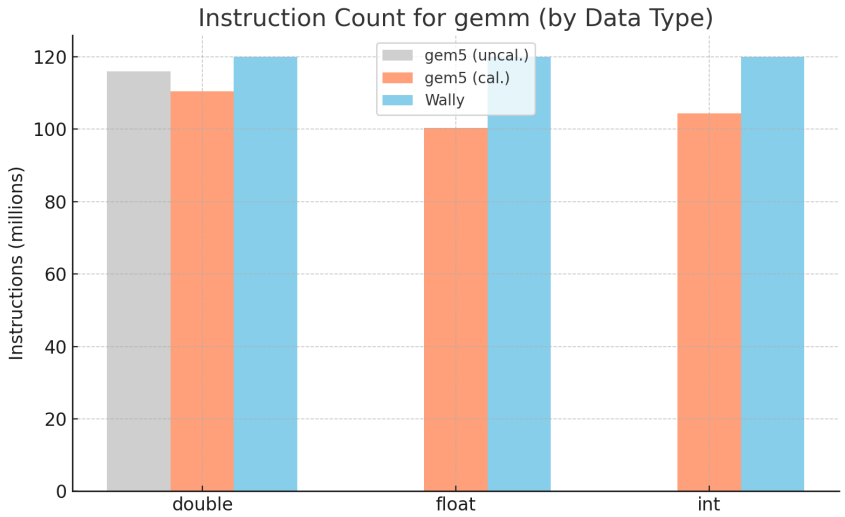


Figure 5.2: Instruction count (in millions) for `gemm` across data types.

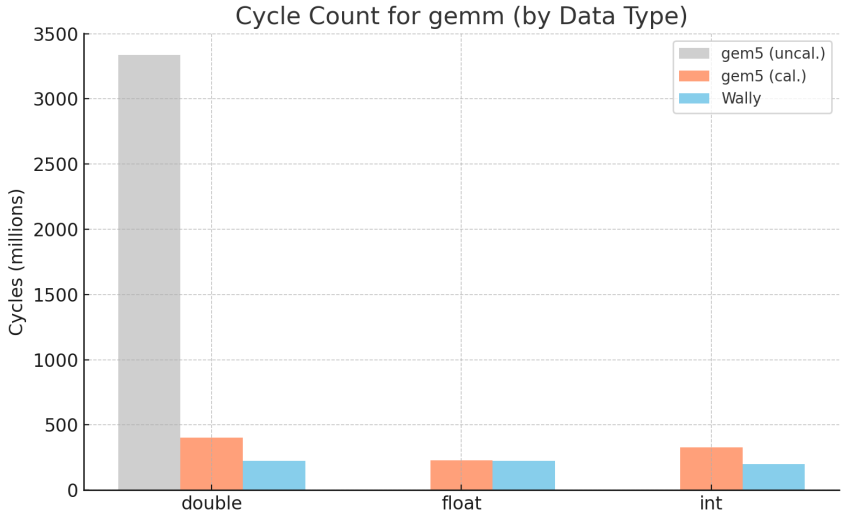


Figure 5.3: Cycle count (in millions) for `gemm` across data types.

5.2. Gemver

The `gemver` benchmark performs rank-1 updates and vector additions. It is memory-bound and useful for testing the interaction between memory hierarchy and functional unit latency.

The calibration results across different data types are shown in Figures 5.5, 5.6, and 5.7, where instruction count, cycle count, and CPI are compared between gem5 (calibrated and uncalibrated) and Wally.

These results confirm that the simulation model—once calibrated—captures the memory-bound behavior of `gemver` with a high degree of accuracy. The CPI trends closely follow those of the FPGA across all data types. For the `double` data type, the CPI deviation remained below 17.6%, reinforcing the model's reliability in memory-

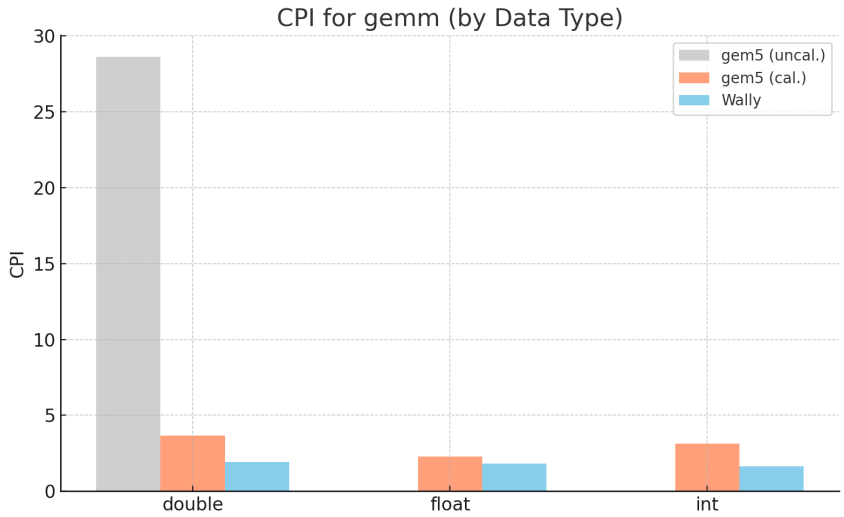


Figure 5.4: CPI for `gemm` across data types.

sensitive scenarios.

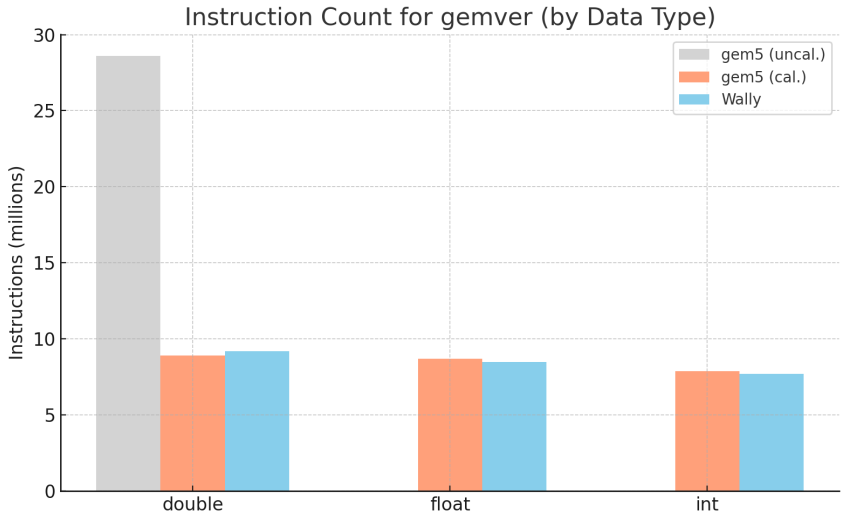


Figure 5.5: Instruction count (in millions) for `gemver` across data types.

5.3. LU

The `lu` benchmark performs LU decomposition of a dense matrix. It is compute-intensive and features nested loops with significant data reuse, making it well-suited for analyzing arithmetic throughput, memory latency, and cache behavior.

Figures 5.8, 5.9, and 5.10 show the instruction count, cycle count, and CPI respectively for different data types.

The `lu` benchmark pushes the compute pipeline more heavily and involves a dense sequence of arithmetic operations. Although CPI alignment is not as tight as

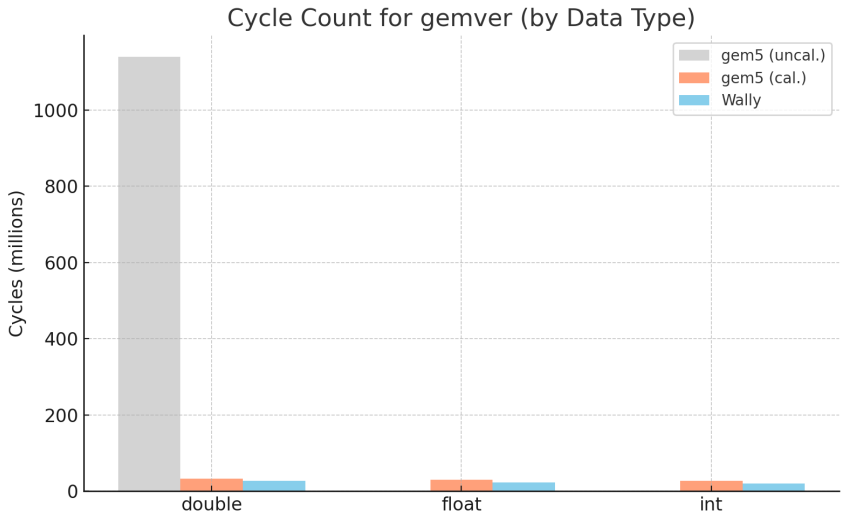


Figure 5.6: Cycle count (in millions) for `gemver` across data types.

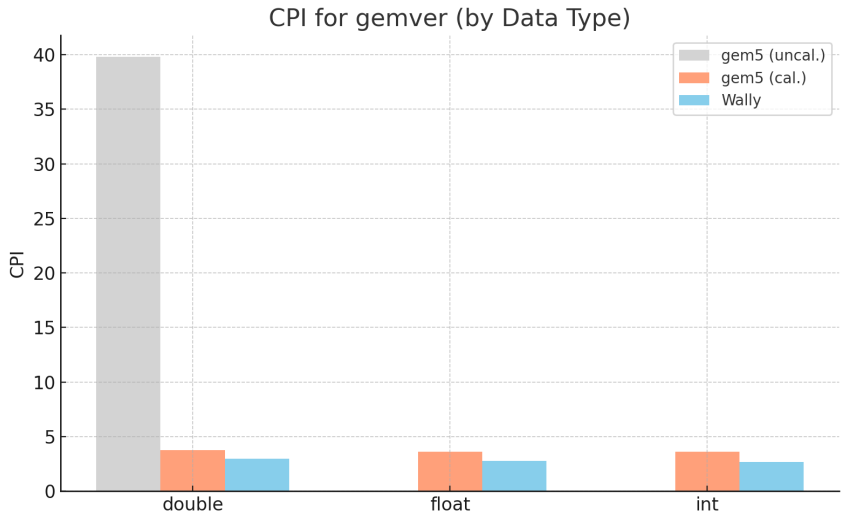


Figure 5.7: CPI for `gemver` across data types.

with simpler kernels, the calibrated simulation still captures the execution trends and instruction behavior reliably. The slight overestimation in simulation is likely due to conservative timing for certain operations, but overall results remain within a justifiable margin. Although the CPI differences here are more pronounced, the simulation model still captures overall execution trends and instruction behavior reliably.

5.4. Symm

The `symm` benchmark performs symmetric matrix-vector multiplication. It features nested loops with moderate reuse patterns, making it useful for analyzing the effects of memory hierarchy and arithmetic throughput in simulation.

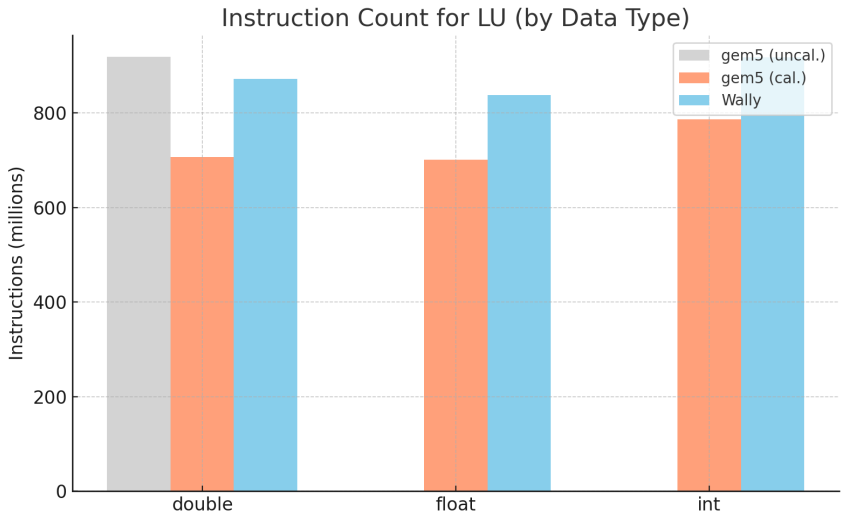


Figure 5.8: Instruction count (in millions) for `lu` across data types.

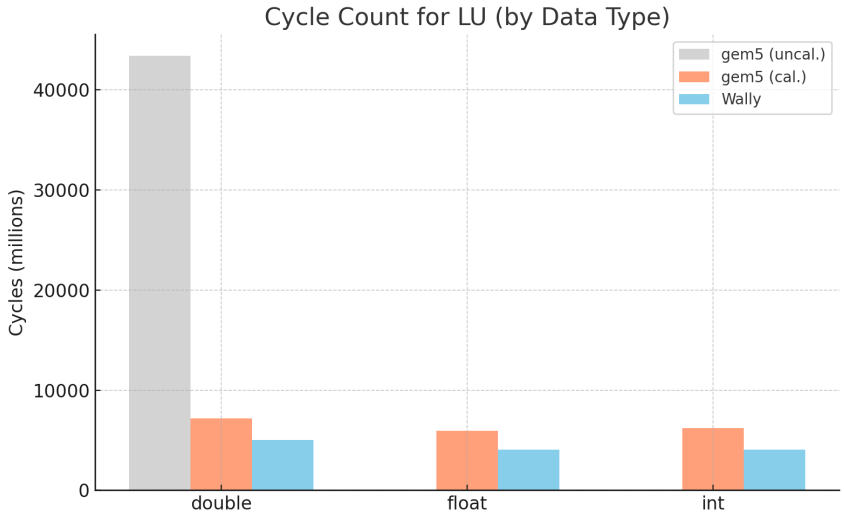


Figure 5.9: Cycle count (in millions) for `lu` across data types.

Figures 5.11, 5.12, and 5.13 show the instruction count, cycle count, and CPI, respectively, across three data types and platforms.

In `symm`, which involves symmetric matrix-vector multiplication, the simulation model performs well in matching the hardware profile. CPI values across all data types remain close to those measured on the FPGA. The accuracy here is notable, given that the benchmark involves a moderate mix of arithmetic and memory access patterns, showing that the calibrated model generalizes well to hybrid workloads.

5.5. Syr2k

The `syr2k` benchmark performs symmetric rank-2k updates. It is representative of memory-bound workloads with moderate reuse and is useful for evaluating latency

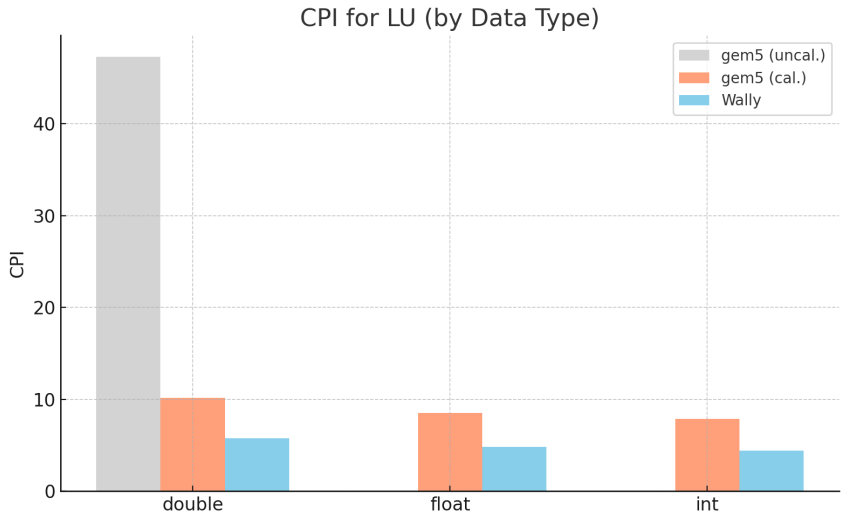


Figure 5.10: CPI for `lu` across data types.

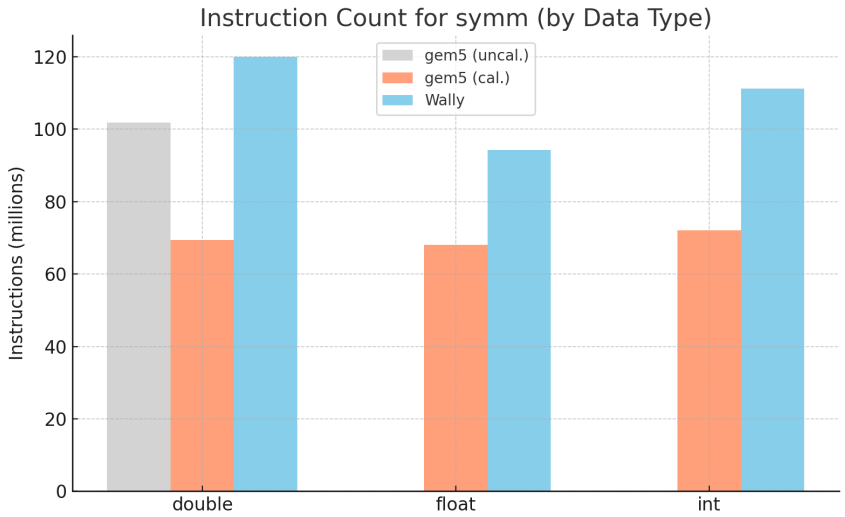


Figure 5.11: Instruction count (in millions) for `symm` across data types.

and cache efficiency.

Figures 5.14, 5.15, and 5.16 show the instruction count, cycle count, and CPI for different data types and simulation configurations.

The `syr2k` benchmark further validates the model's ability to handle memory-sensitive kernels. The CPI deviation was 13.3% for `float` and 18.5% for `double`, indicating strong alignment between simulation and hardware in these configurations. This result highlights the simulator's robustness in modeling cache and memory subsystem interactions.

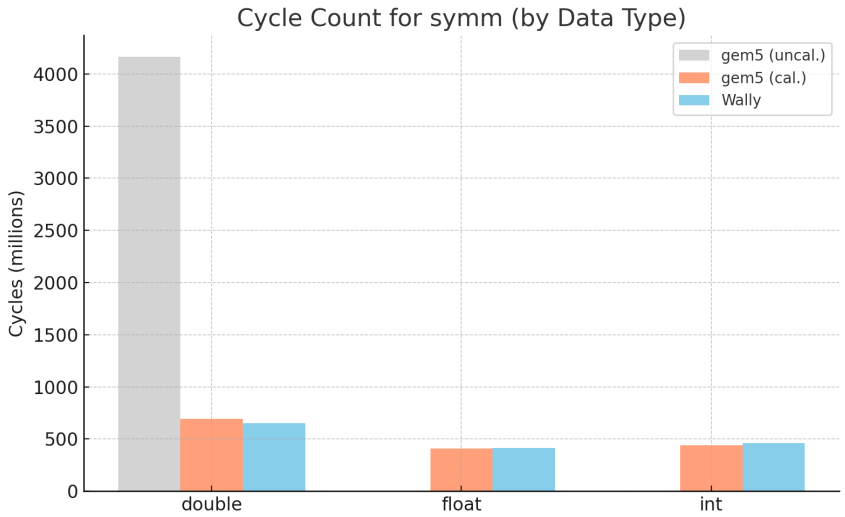


Figure 5.12: Cycle count (in millions) for `symm` across data types.

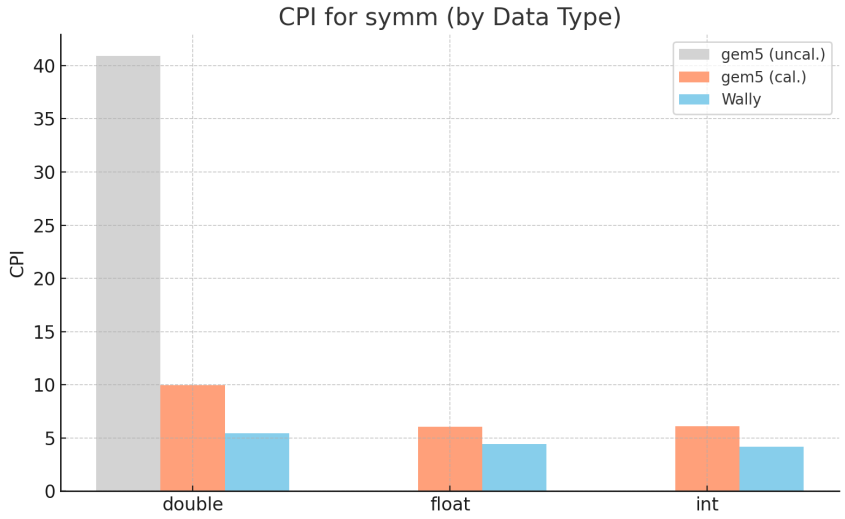


Figure 5.13: CPI for `symm` across data types.

5.6. Final Remarks

To evaluate the accuracy of the calibration process, CPI measurements were collected across five PolyBench benchmarks and compared between gem5 and the Wally FPGA for three data types: `int`, `float`, and `double`. The results are shown in Figures 5.17, 5.18, and 5.19.

From these results, `int` data types yielded the highest simulation accuracy, with some average CPI deviations under 10%. In contrast, `double` operations showed larger discrepancies, likely due to multi-cycle latencies and floating-point pipeline behavior that are more difficult to replicate precisely.

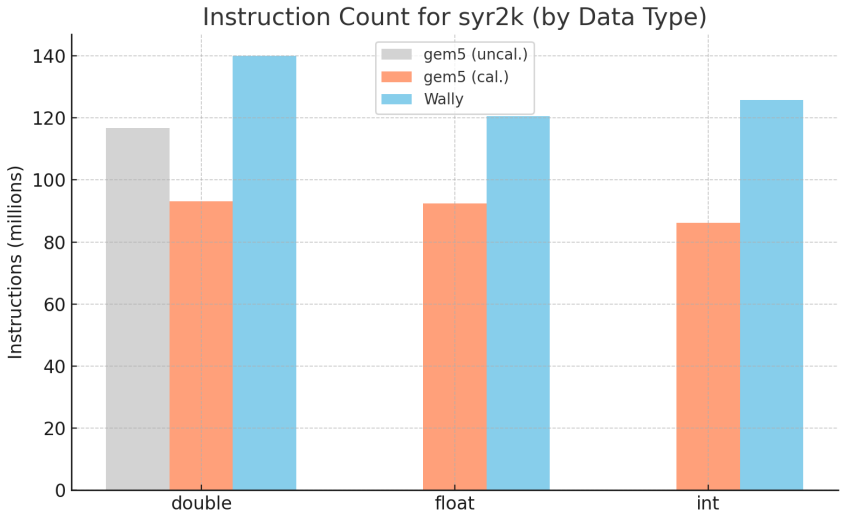


Figure 5.14: Instruction count (in millions) for `syr2k` across data types.

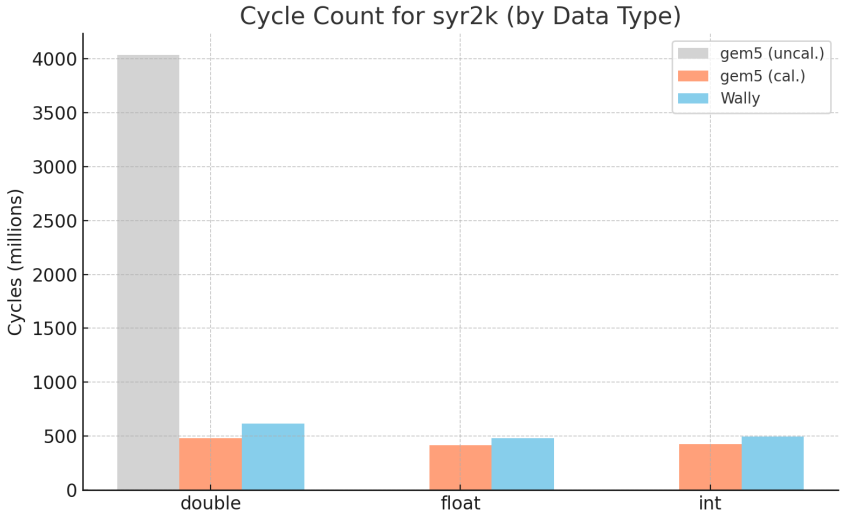


Figure 5.15: Cycle count (in millions) for `syr2k` across data types.

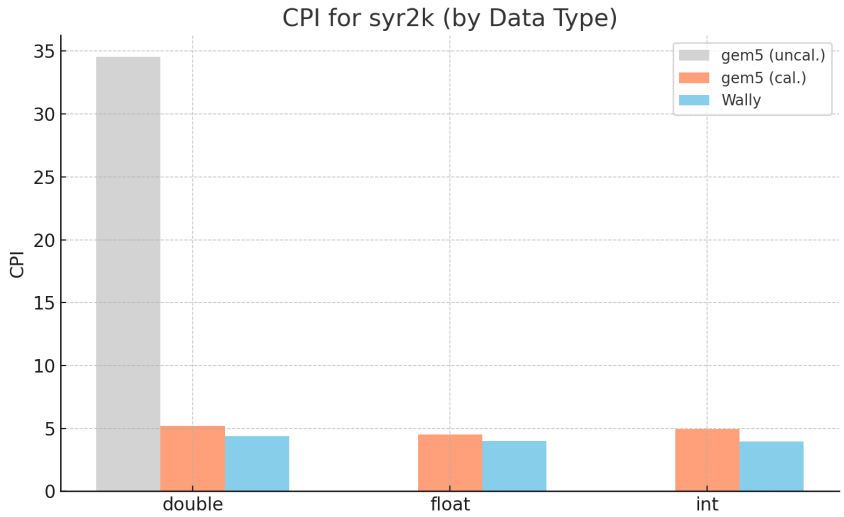


Figure 5.16: CPI for `syr2k` across data types.

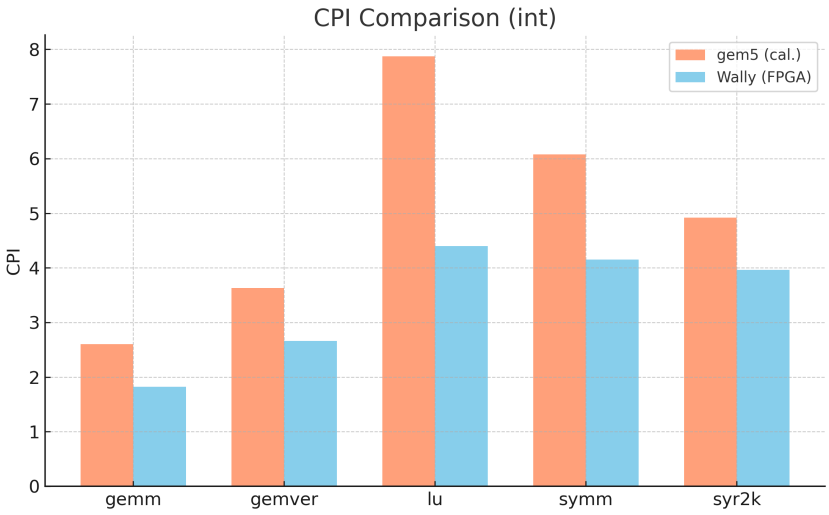


Figure 5.17: CPI comparison between gem5 and Wally for `int` data type.

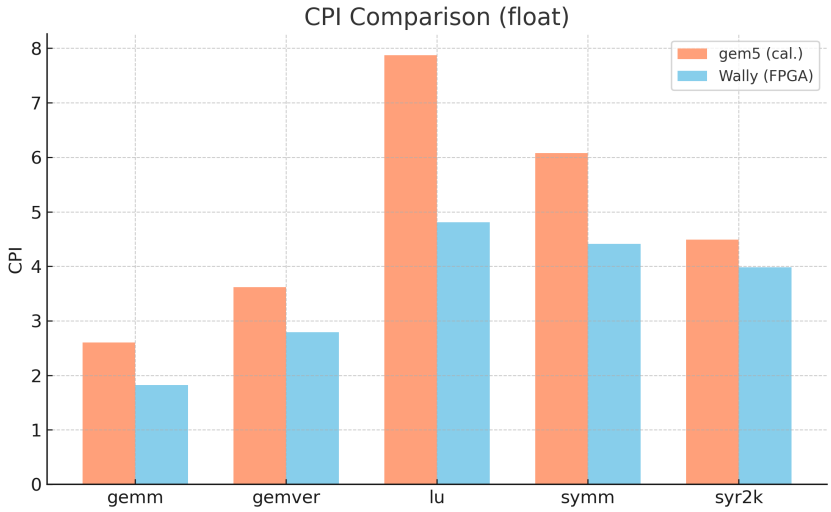


Figure 5.18: CPI comparison between gem5 and Wally for `float` data type.

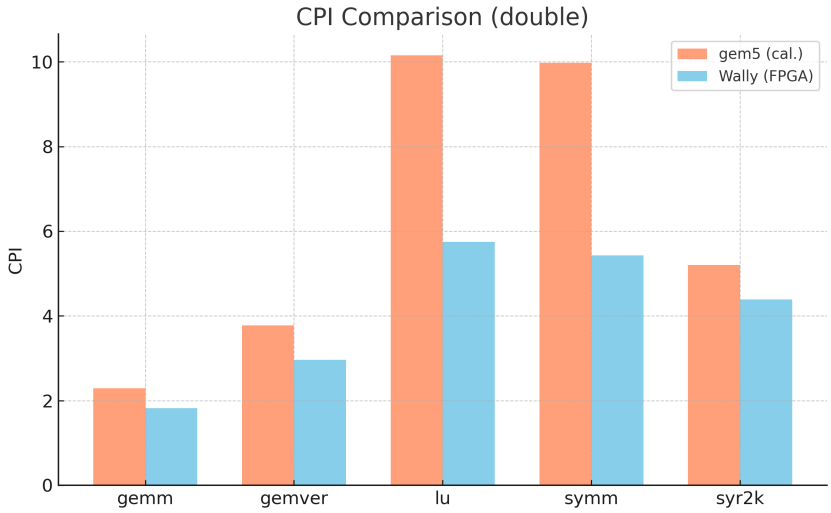


Figure 5.19: CPI comparison between gem5 and Wally for `double` data type.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

The calibration process carried out in this work demonstrates that a cycle-level simulation model in `gem5` can be tuned to closely reflect the behavior of a real RISC-V core implemented on FPGA. By incrementally adjusting architectural parameters, such as cache and memory latency, branch prediction, pipeline widths, and functional unit timing—CPI values in simulation we resulted in a strong alignment with that observed on hardware.

The step-by-step tuning strategy proved highly effective across a range of benchmarks, progressively reducing the simulation error and narrowing the performance gap between gem5 and the FPGA implementation. Integer-based benchmarks showed particularly tight convergence, while floating-point operations exhibited slightly larger but still acceptable deviations. The integration of Verilator traces also played a critical role in fine-tuning specific timing details, especially for multi-cycle operations like multiplication.

Overall, the final simulation model exhibits high reliability across both compute and memory-bound kernels, validating the effectiveness of the calibration process. The resulting configuration can now serve as a robust foundation for architectural analysis, enabling performance evaluation and design exploration without requiring physical access to FPGA hardware.

6.2. Future Work

While the calibration of the CVW-Wally core in `gem5` has yielded accurate simulation results across a wide range of benchmarks, several directions remain open for further development and extension of this work:

- **Expansion of performance counter usage:** The current analysis relies primarily on instruction and cycle counts. Extending this to include more

performance counters—such as cache misses, branch mispredictions, TLB accesses, or stalls—would enable a more detailed and fine-grained calibration of the simulator.

- **Modeling additional hardware components:** The current simulation focuses on the core pipeline and its immediate memory hierarchy. A natural next step is to incorporate more components, such as interrupt controllers, and peripheral buses. Additionally, refining the cache subsystem by tuning parameters such as bus width, number of MSHRs would enable more accurate modeling of system-level interactions and performance bottlenecks.
- **Refining the simulated microarchitecture:** The current gem5 model is based on a generic in-order core (MinorCPU), which has been progressively adapted to resemble the behavior of Wally. A valuable future step would be to further align the model's microarchitectural details, such as forwarding logic, dependency resolution, queue sizes, or pipeline, to more faithfully replicate Wally's actual hardware implementation.
- **Timing-accurate modeling of CVA6:** While CVA6 was deployed on the Genesys2 FPGA for reference purposes, it was not calibrated in gem5. Creating a timing-accurate simulation of CVA6 would enable fairer comparisons.
- **Support for more complex benchmarks:** The calibration was performed using a subset of compute- and memory-bound kernels from the PolyBench suite. Extending the evaluation to include larger applications, such as SPEC or CoreMark, would provide more insights into system-level performance and cache behavior.
- **Research stay and continued collaboration:** As part of the IMEC project, this work will continue during a three-month research stay, where further benchmarks, tooling support, and validation infrastructure will be developed in collaboration with the project team.

These efforts aim to enhance the accuracy and flexibility of the simulation framework, enabling its use in future architectural exploration, performance validation, and RISC-V system design.

Bibliography

Allart, C., Coulon, J.-R., Sintzoff, A., Potin, O. and Rigaud, J.-B. Using a performance model to implement a superscalar cva6. In *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, 43–46. 2024.

Domingos, J. M., Rocha, T., Neves, N., Roma, N., Tomás, P. and Sousa, L. Supporting risc-v performance counters through linux performance analysis tools. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 94–101. IEEE, 2023.

Domingos, J. M., Tomas, P. and Sousa, L. Supporting risc-v performance counters through performance analysis tools for linux (perf). *arXiv preprint arXiv:2112.11767*, 2021.

Huppert, Q., Evenblij, T., Perumkunnil, M., Catthoor, F., Torres, L. and Novo, D. Memory hierarchy calibration based on real hardware in-order cores for accurate simulation. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 707–710. IEEE, 2021.

Kioulos, E. *A Preliminary Accuracy Analysis of Simulated RISC-V Systems*. Diploma work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2023.

Pai, K., Qiu, Z. and Lowe-Power, J. Validating hardware and simpoints with gem5: A risc-v board case study. Poster presented at the gem5 Workshop, International Symposium on Computer Architecture (ISCA), 2023. Accessed: 2025-05-23.

Pathak, K., Klein, J. A. H., Ansaloni, G., Zapater Sancho, M. and Atienza Alonso, D. Validating full-system risc-v simulator: a systematic approach. In *RISC-V Summit Europe 2023*. 2023.

Ravenel, P., Perais, A., de Dinechin, B. and Pétrot, F. A gem5-based cva6 framework for microarchitectural pathfinding. *RISC-V Summit Europe, Jun*, 2023.

Shahid, U., Ahmad, A. and Wasim, S. gem5-based evaluation of cva6 soc: Insights into the architectural design. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 298–300. IEEE, 2024.

Waterman, A., Lee, Y., Patterson, D. A. and Asanovic, K. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, Vol. 116, 1–32, 2011.