TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

A Preliminary Accuracy Analysis of Simulated RISC-V Systems

Author:
Evangelos KIOULOS

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Sotirios IOANNIDIS
Dr. Angelos IOANNOU (FORTH and LBNL)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

December 18, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

A Preliminary Accuracy Analysis of Simulated RISC-V Systems

by Evangelos KIOULOS

The RISC-V Instruction Set Architecture (ISA) maintains a surging interest both in industry and academia due to its simplicity, extensibility, and open license. The integration of RISC-V ISA in the widely used gem5 simulator bridges the gap between RTL and ISA (Spike, QEMU) simulation, as it offers a micro-architectural simulator to the RISC-V ecosystem. This, however, raises uncertainty regarding the accuracy degree of the RISC-V related model implementations in gem5. Especially if we consider that they are still premature due to their recent adoption. The modeling accuracy is crucial as it is responsible for guiding properly research studies and pinpointing areas for optimization on various architectural design spaces. In this thesis, we aim to match the performance and energy costs of an ASIC RISC-V implementation, namely CVA6 (formerly known as Ariane), with a simulated RISC-V system in gem5. We present our experimental setup where we use the gem5 simulator to obtain the performance statistics and McPAT to estimate power and energy metrics. Afterwards, we proceed with an analysis plan to identify potential inaccuracies and flaws of the gem5 simulator. We then evaluate the performance of our simulated system using benchmarks from the RISC-V ecosystem and compare the results to published hardware implementations.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

A Preliminary Accuracy Analysis of Simulated RISC-V Systems

by Evangelos KIOULOS

Η αρχιτεκτονική $RISC-V$ προκαλεί αυξανόμενο ενδιαφέρον, τόσο στη βιομηχανία όσο και στην ακαδημαϊκή κοινότητα, λόγω της επεκτασιμότητας της, του απλού συνόλου εντολών που προσφέρει και του ανοιχτού της κώδικα. Η ενσωμάτωση της αρχιτεκτονικής $RISC-V$ στον ευρέως χρησιμοποιημένο προσομοιωτή $gem5$ γεφυρώνει το χάσμα μεταξύ της RTL προσομοίωσης και της προσομοίωσης επιπέδου συνόλου εντολών (ISA, όπως τα $Spike$, $QEMU$), καθώς εισάγει στο οικοσύστημα της αρχιτεκτονικής $RISC-V$ την προσομοίωση επιπέδου μίκρο-αρχιτεκτονικής. Ωστόσο, η εν λόγω εφαρμογή εγείρει ζητήματα ως προς τον βαθμό ακρίβειας της υλοποίησης της αρχιτεκτονικής $RISC-V$ στο $gem5$, ειδικά αν λάβουμε υπόψη ότι βρίσκεται ακόμα σε πρώιμο στάδιο. Η ακρίβεια στην μοντελοποίηση έχει ιδιαίτερη αξία τόσο στην καθοδήγηση της σχετικής έρευνας, όσο και στην ανάδειξη πεδίων βελτιστοποίησης των διαφόρων αρχιτεκτονικών. Σε αυτή τη διπλωματική εργασία προσπαθούμε να παραγάγουμε την απόδοση και τα κόστη σε ισχύ και ενέργεια μιας $ASIC$ υλοποίησης ενός επεξεργαστή αρχιτεκτονικής $RISC-V$, του $CVA6$ (παλαιότερα γνωστός και ως $Ariane$), σε ένα προσομοιωμένο σύστημα στον προσομοιωτή $gem5$. Παρουσιάζουμε την πειραματική διαδικασία όπου χρησιμοποιούμε τον προσομοιωτή $gem5$ για τον υπολογισμό της απόδοσης του συστήματος και τον προσομοιωτή $McPAT$ για τον υπολογισμό της ισχύος και της ενέργειας. Στη συνέχεια, αξιολογούμε τα αποτελέσματά μας χρησιμοποιώντας βενςημαρκς από το οικοσύστημα της αρχιτεκτονικής $RISC-V$ και συγκρίνοντάς τα με τα δημοσιευμένα.

Acknowledgements

First of all, I would like to thank my supervisors Prof. Apostolos Dollas and Dr. Angelos Ioannou for their support and guidance during this thesis. Assoc. Prof. Sotirios Ioannidis from the thesis committee for his feedback on the text.

I would also like to express my gratitude to Sotiris Totomis for his guidance and insights throughout this thesis. His valuable feedback, help with the tools and comments on the thesis text were vital for the completion of this project.

Last but not least, I would like to thank Prof. Nikolaos Tampouratzis and Dr. Andreas Brokalakis for their helpful comments and advice regarding the use and errors of McPAT.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
API	**A**pplication **P**rogramming **I**nterface
BHT	**B**ranch **H**istory **T**able
BTB	**B**ranch **T**arget **B**uffer
BOOM	**B**erkley **O**ut of **O**rder **M**achine
CSR	**C**ontrol and **S**tatus **R**egisters
CMOS	**C**ompilmentary **M**etal **O**xide **S**emiconductor
CPI	**C**ycles **P**er **I**nstruction
CLINT	**C**ore **L**ocal **Int**errupt Controller
CPU	**C**entral **P**rocessor **U**nit
DDR3	**D**ouble **D**ata **R**ate type 3 memory
DMIPS	**D**hrystone **M**illion **I**nstructions **P**er **S**econd
DSL	**D**omain **S**pecific **L**anguage
DVFS	**D**ynamic **V**oltage **F**requency **S**caling
DTB	**D**evice **T**ree **B**lob
ELF	**E**xecutable and **L**inkable **F**ormat
FS	**F**ull **S**ystem
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FU	**F**unctional **U**nit
FPU	**F**loating **P**oint **U**nit
FD-SOI	**F**ully **D**epleted-**S**ilicon **O**n **I**nsulator
FIFO	**F**irst **I**n **F**irst **O**ut
FESVR	**F**ront-**E**nd **S**er**v**e**r**
GCC	**G**NU **C**ompiler **C**collection
IPC	**I**nstructions **P**er **C**ycle
ISA	**I**nstruction **S**et **A**rchitecture
ITRS	**I**nternational **T**echnology **R**oadmap for **S**emiconductors
KVM	**K**ernel-based **V**irtual **M**achine
LSU	**L**oad **S**tore **U**nit
MMU	**M**emory **M**anagement **U**nit

MIPS	**M**illion **I**nstructions **P**er **S**econd
OS	**O**perating **S**ystem
O3-OOO	**O**ut-**O**f-**O**rder
PC	**P**rogram **C**ounter
PTW	**P**age **T**able **W**alker
PMA	**P**hysical **M**emory **A**ttribute
PLIC	**P**latform-**L**evel **I**nterrupt **C**ontroller
RISC	**R**educed **I**nstruction **S**et **C**omputer
ROB	**R**e**o**rder **B**uffer
RTL	**R**egister **T**ransfer **L**evel
RAS	**R**eturn **A**ddress **S**tack
RAM	**R**andom **A**ccess **M**emory
SoC	**S**ystem-**o**n-**C**hip
SE	**S**ystemcall **E**mulation
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
TDP	**T**hermal **D**esign **P**ower
TLB	**T**ranslation **L**ookaside **B**uffer
TVM	**T**est **V**irtual **M**achine
VLIW	**V**ery **L**ong **I**nstruction **W**ord

Dedicated to my family and friends…

Chapter 1

Introduction

1.1 Motivation

With the demand for computer chips growing, the design and simulation can become a complex, costly, and time-consuming process. There are different approaches when designing computer hardware. Designers, usually use hardware description languages (HDL), such as VHDL or SystemVerilog, to implement their architectures and simulate them using register-transfer level (RTL) simulators. RTL simulators can simulate real hardware designs in great detail and cycle accuracy and also provide accurate estimations on power consumption and hardware area. However, implementing the designs can be a complex process that limits design space exploration, and simulations can be very time-consuming, even taking up to days, when simulating real-world applications, depending on the complexity of the simulated system. Another approach is to run their RTL designs on FPGAs. The simulation times can be orders of magnitude faster compared with the RTL simulators due to the execution taking place on real hardware. Nonetheless, implementation of the designs and acquisition of performance statistics can be a complex task, and investing in FPGA hardware can be costly. Furthermore, in-depth testing demands tailored OS support with its associated software development effort across the stack.

Micro-architectural simulators offer faster simulations, compared to RTL simulators, by modeling hardware modules at higher levels of abstraction. This, allows designers to easily simulate and measure the performance of various architectures at relatively fast simulation speeds, making them great for design space exploration. Also, most of these architectural simulators offer complete OS support, thus their use eliminates software development-related costs. Their capability to simulate both hardware and OS implementations and to measure their interaction, makes them suitable for software-hardware co-design exploration, too. Notably, such simulators are widely used as implementation and measurement tools by the research community. However, high-level abstraction models can lead to a great loss of accuracy, because they omit low-level details that can affect performance. The gem5 Simulator is a micro-architectural simulator that has gained popularity in both academia and industry. gem5 offers a great variety of instruction set architectures and highly configurable hardware models, an easy-to-use front-end interface and the capability to simulate a full operating system. Furthermore, gem5's open-source license makes it accessible to researchers, students and the industry.

Despite gem5 being open source, most of the ISAs supported have proprietary licenses. A solution to this comes with the addition of the RISC-V ISA to gem5. RISC-V offers an open and simple instruction set that can implement numerous architectures, from low-cost embedded to high-end processors, through its ISA extensions. The addition of RISC-V to gem5, offers an alternative simulation framework for RISC-V, bridging the gap between RTL simulation and ISA simulation with simulators such as Spike [1]. *With the growing popularity of RISC-V, the accuracy of gem5's models in the context of RISC-V becomes of great importance. Especially, if we consider that the RISC-V related implementations in the gem5 simulator are still premature.*

1.2 Objectives

This work focuses on the issue of micro-architectural simulator accuracy, specifically for the recently added RISC-V framework of gem5. The main objective of this thesis is to match the energy costs and performance of an ASIC implementation of a RISC-V CPU, particularly Ariane, with a simulated RISC-V system and point out weaknesses and possible accuracy loss sources of the simulator.

First of all, the author developed configuration scripts on gem5's front-end that set up the simulated system and run the simulation, on gem5's Full-System and bare-metal modes. In addition, the author developed scripts that model Ariane's pipeline and main architectural modules, using gem5's detailed CPU models, that can either be combined with the configuration scripts developed by the author or gem5's default configuration scripts with minimal modifications, to simplify the configuration and calibration of the system's micro-architectural design parameters. Furthermore, in order to run in Full-System mode, the author used a RISC-V built Linux kernel along with a compatible disk image.

Since gem5 does not produce power and energy statistics, the author used scripts that integrate gem5 with a power simulator, and in particular McPAT. Specifically, the author used scripts that use gem5's output files along with an input template and produce an input file that can be used by McPAT. Moreover, the author modified the template XML file in order to match Ariane's specifications and enable it to be used with gem5's latest version, specifically in this study, version 21.2.1.

To evaluate the performance of the simulated system, the author created a baseline configuration of the simulated system and simulated benchmarks from the *riscv-tests* suit on the base configuration and on multiple configurations of the base model with calibrated micro-architectural design parameters. Particularly, the author selected several tests and benchmarks from the suite that match the workload described in [2], modified the assembly tests to run longer in order to obtain more meaningful power and performance statistics and ported them in order to be used with the gem5 simulator, using the cross-compilers of the RISC-V toolchain. Last but not least, the author compared the results of the simulation to the published performance, power and energy statistics and analyzed the possible causes of accuracy loss in the gem5 simulator.

1.2.1 Contributions

This study offers an early-stage analysis of possible accuracy errors of the gem5 simulator in the context of the recent RISC-V related implementations. It provides a starting point for further research on the support of RISC-V in gem5 pointing out weaknesses of the simulator.

Furthermore, this project provides an in-depth guide on development with gem5's RISC-V framework on bare-metal and Full-System simulation, showcasing possible challenges during development. Specifically, it offers details on the use and implementation of configuration scripts on gem5's front-end, that setup and the simulation of the target system, utilizing gem5's detailed models. In addition, we present the utilization of the *gem5-resources* library and the compatibility of the RISC-V toolchain with gem5. Moreover, we demonstrate the integration of gem5 with external tools, in this case, McPAT, along with the challenges that occur.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** provides a theoretical basis on the RISC-V instruction set architecture and the RISC-V software and hardware ecosystem, the RISC-V core used in this study (CVA6 Ariane), the benchmarks and the tools used, gem5 and McPAT.
- **Chapter 3 - Related Work:** explores related work on micro-architectural simulation and accuracy evaluation using gem5 and McPAT. Specifically, it presents previous work on the accuracy evaluation of the gem5 simulator in the context of the ARM instruction set, the implementation, functional validation and accuracy evaluation of the RISC-V instruction set in gem5, the integration of gem5 with external tools and McPAT and accuracy evaluation of the McPAT power simulation.
- **Chapter 4 - Simulation Environment Setup and Implementation:** describes the experimental environment setup and the simulation framework, providing detail on the implementation of the configuration scripts, the setup and the modifications on the benchmarks used, the implementation and the micro-architectural design parameter selection for the baseline model.
- **Chapter 5 - Results and Performance Analysis:** describes the calibration of the micro-architectural design parameters of the baseline model and the parameter selection for the calibrated configurations, the experiments conducted in order to obtain the performance, power and energy statistics and comparison between the results and the published statistics.
- **Chapter 6 - Conclusions and Related Work:** draws conclusions and gives directions for future work.

Chapter 2

Theoretical Background

2.1 RISC-V Architecture and Simulation Environments

2.1.1 RISC-V Instruction Set Architecture

RISC-V is a freely licensed and open standard instruction set architecture, introduced by the University of California, Berkeley in 2010. It offers a simple load-store instruction set with principals similar to other RISC architectures. It has gained popularity both in industry and academia through its simplicity and its improvements compared to other open ISAs [3]. RISC-V provides both 32-bit and 64-bit instruction sets along with various extensions (e.g. compressed instructions, vector operations, and more that are ongoing work at the time of writing this thesis). The base instruction set is split into user-level and privileged instructions, with the latter enabling additional functionality required for running an operating system.

RISC-V Instruction set organization

The RISC-V ISA consists of three base instruction sets that support 32 and 64-bit registers and a variety of extensions that can be added to one of the base instruction sets. This allows RISC-V to be implemented in a wide range of applications from low-cost embedded processors to high-end processors with sophisticated designs and multi-core configurations. A common form of the base ISA with standardized extensions is *RV64IMAFD*, which implements the RV64I base integer ISA with the M extension for multiply/divide support, A extension for atomic memory operations and F for single and D for double precision IEEE floating point support. This is often abbreviated as *RV64G*, G meaning general purpose, with *RV32G* being the 32-bit subset. Table 2.1 lists base instruction sets and standard extensions with their descriptions.

TABLE 2.1: RISC-V base ISA and extensions.

Name	**Functionality**
RV32I	Base 32-bit integer instruction set, with 32 registers
RV32E	Base 32-bit integer instruction set(embedded), with 16 registers
RV64I	Base 64-bit integer instruction set
RV128I	Base 128-bit integer instruction set
M	Adds integer multiply and divide instructions
A	Adds support for atomic instructions
F	Adds single precision (32-bit) IEEE floating point
D	Extends floating point to double precision
Q	Further extends floating point to add support for quad precision
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a 16-bit compressed version of the instruction set
B	Standard extension for bit manipulation
J	Adds support for Dynamically Translated Languages
T	Adds support for Transactional Memory
P	Adds Packed-SIMD instruction support
V	Adds support for Vector operations
N	Standard extension for user-level interrupts
H	Standard extension for Hypervisor
S	Standard extension for Supervisor-level instructions

RISC-V Registers, Data structures and Addressing modes

RV64G uses 32 64-bit general-purpose registers, *x0, x1,...,x31*, also known as *integer* registers, with x0 being the *zero* register. The supported data types for integer data are 8-bit bytes, 16-bit half-words, 32-bit words and 64-bit for double-words. Half-words are included as they are commonly found in languages like C and are popular in programs, such as operating systems. Bytes, half-words and words are loaded to the general-purpose registers with either zeros or the sign bit repeated to fill the 64 bits of the register.

The F and D extensions add 32 floating point registers, *f0, f1,...,f31*, that can hold 32-bit single precision or 64-bit double precision values. When holding single-precision values, the upper half of the register is unused. Operations for single and double-precision floating-point are provided.

There are two data addressing modes, immediate and displacement, both with 12-bit fields. Placing 0 in the 12-bit field enables register indirect mode, while using the *zero* register as a base register enables limited absolute addressing. The RV64G memory is byte addressable with a 64-bit address and uses little-endian byte numbering. RISC-V is a load-store architecture and

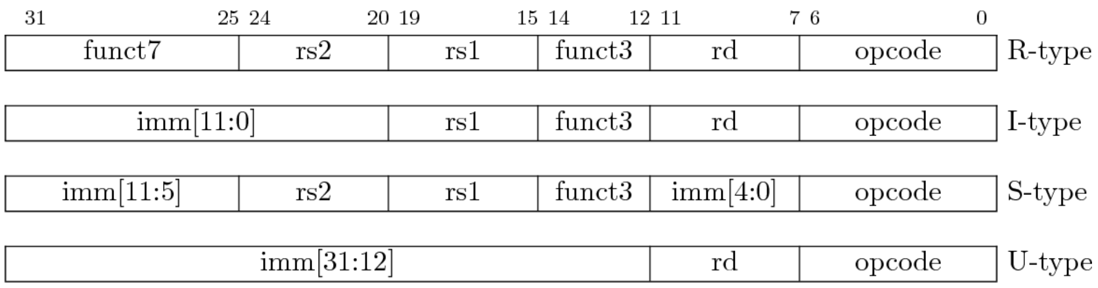


FIGURE 2.1: RISC-V Instruction Formats, taken from [5].

all references between the memory and the general-purpose or floating point registers are through load-stores. Memory accesses don't have to be aligned, however, unaligned accesses are extremely slow [4].

RISC-V Instruction Formats

There are four core instruction formats R, I, S and U, as shown in Figure 2.1, all of which have a fixed length of 32-bits with a 7-bit opcode. All instructions must be aligned on a 4-byte boundary memory. A misaligned taken branch or unconditional jump can generate an *instruction address misaligned* exception if the target address is not 4-byte aligned.

The source (rs1 and rs2) and destination (rd) registers are in the same positions throughout all instruction formats except for the 5-bit immediate Control/Status Register (CSR) instructions. Immediate values are always sign extended and the sign bit is always bit 32. The *opcode* specifies the instruction type while the *funct* specifies the specific operation. Several formats can encode multiple types of operations, such as the I-format which can encode both ALU immediate and load instructions and the S-format for stores and conditional branches.

There are two additional formats, B and J, that are used based on the handling of the immediate. The B format is a variant of the S format, written also as SB, and it is used for conditional branches, and the J format is a variant of the U format, written also as UJ, used for jump instructions. Figure 2.2 shows all RISC-V instruction formats along with SB and UJ.

RISC-V Privilege modes

The privileged instruction set is independent from the user-level ISA and it includes different levels of hardware support needed to run an operating system.

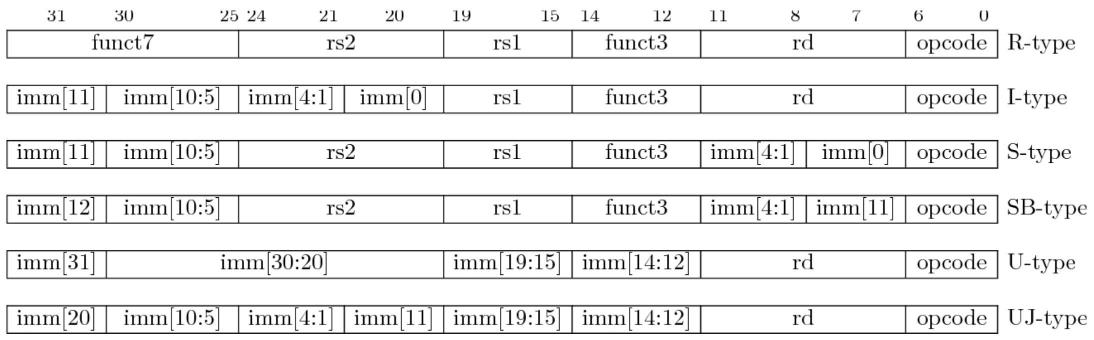

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11] | imm[10:5] | imm[4:1] | imm[0] | rs1 | funct3 | rd | opcode | I-type
imm[11] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | SB-type
imm[31] | imm[30:20] | imm[19:15] | imm[14:12] | rd | opcode | U-type
imm[20] | imm[10:5] | imm[4:1] | imm[11] | imm[19:15] | imm[14:12] | rd | opcode | UJ-type

FIGURE 2.2: RISC-V Instruction Formats including SB and UJ, taken from [5].

There are four currently available main privilege levels, user, supervisor, hypervisor and machine. The privilege levels provide protection between components in the software stack. Operations that are not permitted in a specific privilege level can raise an exception, which is handled by a trap.

TABLE 2.2: RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Hypervisor	H
3	11	Machine	M

Machine level (M-mode) has the highest privileges and it is the only mandatory level required in an RISC-V machine. Code that runs in machine mode is trusted and it has low-level access to the machine implementation. The M-mode is used to manage secure execution environments in RISC-V systems. All implementations must provide M-mode, the simplest implementation can provide only M-mode, however, this implementation lacks protection.

User mode (U-mode) and supervisor mode (S-mode) are intended for conventional applications and operating system usage respectively. Hypervisor mode (H-mode) is intended for virtual machine monitors. S-mode provides isolation between the operating system and the supervisor execution environment and hardware abstraction layer code, while H-mode provides isolation between the virtual machine monitor and hypervisor execution environment and hardware abstraction layer running in machine mode.

A hardware thread runs in U-mode until a trap forces a switch to a trap handler. The thread will execute the trap handler and resume at or after the original instruction that caused the trap in U-mode. Traps that increase the privilege level are called *vertical traps*, while traps that remain at the same level are called *horizontal traps*.

Implementations may include a debug mode (D-mode) that supports off-chip debugging and manufacturing tests. D-mode gives even more access than M-mode, thus, it can be considered an additional privilege mode. It may reserve CSR addresses and portions of physical memory.

RISC-V Software Ecosystem

RISC-V software compilation is supported through *GCC* and *LLVM*, with the *RV32GC* and *RV64GC* instruction set extensions being implemented and the *RV32E* base ISA supported only on *GCC*. The GNU C library, *Glibc*, along with the *Newlibc*, which is the C standard library implementation for embedded systems, are supported. Furthermore, the GNU *Binutils*, which is a collection of binary tools, is supported and debugging is provided via the GDB (GNU Debugger). Although the base instruction sets and standard extensions have been implemented, they have not yet been fully optimized.

Emulation of RISC-V systems can be achieved using *QEMU* [6]. QEMU is a free and open-source machine emulator and can support full-system emulation for RISC-V systems. It supports both 32 and 64-bit base instruction sets and standard extensions, along with privileged and unprivileged instruction specifications. Furthermore, RISC-V systems can be simulated using *Spike* [1]. Spike is the RISC-V ISA simulator and implements a functional model. It supports the latest versions of privileged and unprivileged instruction specifications and all base RISC-V instruction sets along multiple standard and non-standard extensions. This enables Spike to be used as a reference model. Spike can simulate RISC-V systems with one or more hardware threads.

2.1.2 CVA6 (former Ariane)

CVA6 is an application-class 64-bit RISC-V CPU core, fabricated under *GlobalFoundries* 22nm FD-SOI technology [2]. The core runs at 1.7GHz and, as they report [2], achieves an efficiency of up to 40Gop/sW. Ariane implements the *RV64GC* ISA and has a 6-stage, single issue, in-order pipeline with full hardware support for multiply/divide, atomic memory operations as well as an IEEE-compliant floating point unit. It supports the compressed as well as the full privileged instruction set extensions. The instructions are issued in-order and executed out-of-order. The core's front-end consists of a *PC generation* stage and the instruction *Fetch* stage, while the back-end consists of the *Decode*, instruction *Issue*, *Execute* and *Commit* stages. Figure 2.3 shows Ariane's 6-stage pipeline.

The *PC generation* stage is responsible for selecting the next PC. The next PC can originate from a return from an environment call, an interrupt or exception, a branch prediction or a falsely predicted branch, a pipeline flush due to CSR side effects, the debug interface, or a consecutive fetch. The *Fetch* stage contains fetch and pre-decode logic, the instruction cache, and the branch prediction units. It receives information from the PC stage, asks the MMU for an address translation on the requested PC and controls the instruction cache. The data coming from the instruction cache, are registered before being pre-decoded. As a result, a cycle is lost, even on a correct control flow prediction, as the next PC cannot be calculated in the same cycle that the data from the instruction cache is received. They report that, due to the compressed instruction set extension, they fetch on average 1.5 instructions, thus, the lost cycle does not cause a problem. The pipeline's front-end is fully decoupled from the back-end using the instruction queue, which is implemented as a FIFO of configurable depth. The instruction queue stores the instructions in compressed form.

The first stage of the core's back-end is the *Decode* stage. The decode stage re-aligns potentially unaligned instructions, decompresses them and decodes them, The decoded instructions are then put in an issue queue. The *Issue* stage houses the issue queue, the scoreboard and the Reorder Buffer (ROB). When the operands are ready the instructions are issued in the execute stage. The *Execute* stage contains all the functional units. Ariane has six functional units, the ALU, a multiplier/divider, a CSR buffer, a Branch Unit, a Load-/Store unit (LSU) and a floating point unit (FPU). Every functional unit is handshaken and readiness is taken into account during the instruction issue. The instructions can retire out-of-order from the functional units and write-back conflicts are resolved through the ROB. Finally, the *Commit* stage reads from the ROB and commits all instructions in program order and the register file is updated. The commit stage can commit two instructions per cycle.

Ariane's main units and key features:

- **Branch Prediction:** A Branch History Table (BHT) with 8 entries and a 2-bit saturating counter, a Branch Target Buffer (BTB) with 8 entries and a Return Address Stack (RAS).

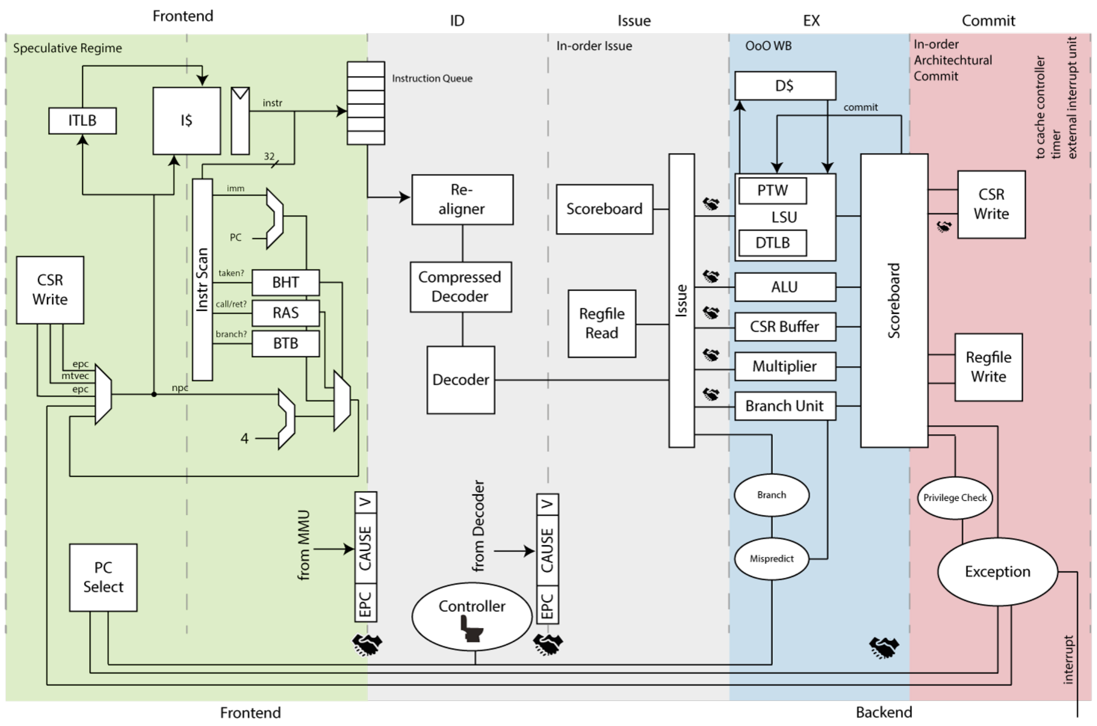


FIGURE 2.3: Ariane's 6-stage Pipeline.

- **Virtual Memory:** Full hardware support for address translation via the Memory Management Unit (MMU). The MMU contains data and instruction TLBs with 16 entries each as well as a Page Table Walker (PTW). Ariane implements a 39-bit, page-based virtual memory scheme, *SV39*.
- **Register Files:** Two physically different register files for integer and floating-point registers, containing 32 64-bit registers.
- **Scoreboard/Re-order Buffer:** Implemented as a circular buffer, contains issued, decoded, in-flight instructions and speculative results written back by various FUs.
- **Caches:** 16kB 4-way instruction cache with a latency of 1 cycle, 32kB 8-way data cache with a latency of 3 cycles. Both data and instruction caches are virtually indexed and physically tagged and include an additional pipeline stage on their outputs. The data cache is a write-back cache and supports hit under miss functionality.
- **Memory and Control Interfaces:** Advanced eXtensible Interface (AXI) 5 master port and four interrupt sources, Machine External Interrupts, Supervisor External Interrupts, Machine Timer Interrupts, Machine Software Interrupts.

- **Functional Units:**
  - **ALU:** Covers most of the base RISC-V ISA, including branch target calculations.
  - **Load/Store Unit (LSU):** Integer and floating-point load/stores as well as atomic memory operations.
  - **Floating-point Unit (FPU):** IEEE compliant floating-point unit with custom trans-precision extensions.
  - **Branch Unit:** An extension to the ALU, handles branch-prediction and branch-correction.
  - **CSR:** RISC-V mandates atomic operations on its CSR, corresponding write data is buffered in this unit and read again when the instruction retires.
  - **Multiplier/Divider:** Fully pipelined 2-stage multiplier and a bit-serial divider with input preparation, takes 2-64 cycles depending on the operands.

Ariane supports the privileged ISA specification and implements user, supervisor and machine privilege levels as well as a RISC-V compliant *Debug* interface. The debug interface requires one additional instruction, *dret*, to return from debug mode. The communication with the external debugger is done through a debug module peripheral.

The core has been taped-out in *GlobalFoundries* 22 FDX technology node and uses a shared System on Chip(SoC) for off-chip communication. Ariane can communicate with the SoC via a full-duplex 64-bit AXI interconnect. The SoC contains 520kB of on-chip scratchpad memory, HyperRAM, SPI, UART and I2C. The core is separately clocked, supplied and powered and the logic cells can be forward body biased to increase speed at the expense of leakage power. The design has been signed-off at 902 MHz at 0.72 V, 125 C, SSG with the final netlist containing 75.34% low voltage threshold and 24.66% super low voltage threshold cells.

With this architecture, they report up to 1.65 DMIPS/MHz depending on the branch-prediction configuration and load latency and an IPC of 0.82 on the *Dhrystone* benchmark with a 128-entry BHT and 64-entry BTB configuration. They report a total leakage of 1.08 mW and a total energy of 51.80 pJ for a generalized matrix multiplication.

A version of Ariane has also been fabricated with *Ara* [7] in *GlobalFoundries* 22 FDX technology. Ara is a 64-bit vector co-processor that implements RISC-V's vector extension. It runs at more than 1GHz and achieves a performance of 33 DP-GFLOPS and energy efficiency of 41 DP-GFLOPS/W.

2.1.3 Alternative RISC-V Hardware

Since the RISC-V ISA was introduced, there have been multiple cores and SoCs that implement the RISC-V architecture. [8] lists several RISC-V cores and SoCs along with their status.

A notable platform widely used in research is the *Rocket Chip generator* [9], developed at UC Berkeley. The Rocket Chip generator is an SoC generator, written in Chisel HDL [10], that uses configurable chip-building libraries for constructing RISC-V based SoCs. It provides a core generator, with optional FPUs, configurable FU pipelines and branch predictors, cache and TLB generators with configurable size, associativity and replacement policies and a generator for peripherals. The generator supports multicore configurations, cache-coherence and heterogeneity through the tile generator and a Rocket Custom Co-processor interface (*RoCC*) for application-specific co-processors. It uses the Rocket Core by default and can also be configured to use the BOOM out-of-order core.

The *Rocket Core* is a 5-stage in-order scalar core that implements the RV32G and RV64G instruction sets, written in Chisel HDL. Rocket has a non-blocking data cache and supports branch prediction through a BHT, a BTB and a RAS in its instruction fetch stage. The core uses the Chisel floating-point implementations for its' floating point units. Address translation is supported by the MMU, with a page-based virtual memory, and supports the RISC-V machine, supervisor and user privilege levels. The Rocket core can also be used as a component library, and many of Rocket's modules can be used in different designs. The Rocket Chip, and Rocket Core, have been taped out multiple times and produced functional silicon chips capable of running Linux.

The *Berkley Out-of-Order Machine (BOOM)* [11] is an open-source synthesizable and parametric out-of-order superscalar RISC-V core that implements the RV64GC instruction set. The first version of BOOM was used for educational purposes, it was similar to the *MIPS R10k* core and featured a simple pipeline with a unified register file. The second version, *BOOM v2* [12], expanded the front-end pipeline stages, modified the branch prediction units, separated the floating-point into an independent pipeline and split the issue queue into separate queues for integer, memory and floating-point operations. BOOM v2 was fabricated in the *BROOM* chip [13], in 28 nm TSMC technology. The third version of BOOM, *SonicBOOM* [14], improved upon bottlenecks of the previous versions, modifying the instruction fetch unit, execution back-end and load/store unit, and added a high performance *TAGE* branch predictor. Furthermore, they added support for the RISC-V compressed instruction set. BOOM was written in Chisel HDL and adopts components from the in-order *Rocket Core*, such as the MMU, L1 caches and execution units.

2.2 Benchmarks

Benchmarks are computer programs used to measure, compare and evaluate the relative performance of a computer system. There are different types of benchmarks such as kernels, small programs that include key aspects of real-world programs, or toy programs, however, they could be characterized into two main categories: synthetic benchmarks and application ("real-world") based. Synthetic benchmarks are fake programs, usually small in size, that try to mimic the behavior of real-world applications, often used to measure or debug specific features of a system. Application based benchmarks use software from real applications and are used to measure the overall performance of a system. Usually, they have large code and data storage requirements.

Collections of benchmark applications are called *benchmark suites*, and they are a popular measure of the performance of processors with a variety of applications. A key advantage of benchmark suites is that the weakness of any benchmark is lessened by the presence of the others. [4] Popular benchmarks and benchmark suites commonly used in academia and the industry include the *Standard Performance Evaluation Corporation* (SPEC) suite, with its' current version being the *SPEC2017*, the EEMBC CoreMark benchmark and CoreMark Pro suite, the PARSEC suite and many more.

For this project, based on the reported workload used in [2], we used tests from the *riscv-tests* suite and the *Dhrystone* benchmark, to match the reported energy costs and performance of Ariane, respectively.

2.2.1 RISCV-Tests suite

RISC-V tests is an extensive assembly test suite in the open-source RISC-V toolchain. The test suite is designed to run on bare-metal machines without any OS support and communicates to a host machine via a front-end server (FESVR) to inform the results. It consists of ISA compliance tests and low-level C tests. The tests are built on various *Test Virtual Machines (TVM)* and can run on different target environments depending on the number of cores and type of memory on the system.

TABLE 2.3: List of RISC-V defined TVMs.

TVM Name	**Description**
rv32ui	RV32 user-level, integer only
rv32si	RV32 supervisor-level, integer only
rv64ui	RV64 user-level, integer only
rv64uf	RV64 user-level, integer and floating-point
rv64uv	RV64 user-level, integer, floating-point, and vector
rv64si	RV64 supervisor-level, integer only
rv64sv	RV64 supervisor-level, integer and vector

Each test uses only features of a given *TVM*. Table 2.1 lists the TVMs currently defined for RISC-V, all of which only support a single hardware thread. These features are defined as the set of registers and instructions that can be used, the portions of memory that can be accessed, the way execution starts and ends and the test data input and output. Each test is contained within a single assembly file which is passed through the C pre-processor and should include the *riscv_test.h* header file which defines macros used by the given *TVM*. This header file differs depending on the target environment. Table 2.2 shows all available target environments. All assembly directives can be used in the test file.

TABLE 2.4: List of available target environments.

Target Env.	**Description**
p	physical memory only, only core 0 boots up
pm	physical memory only, all cores boot up
pt	physical memory only, timer interrupt fires every 100 cycles
v	virtual memory is enabled, only core 0 boots up

Each test program should include the appropriate TVM macro, which specifies the TVM that the test is built on. The format of the macro that defines the possible TVMs is: *RVTEST_RV[32/64]U(F/V)* or *RVTEST_RV[32/64]S*. The 32 or 64 defines 32 or 64-bit instruction decoding, the U is for user space while S is for supervisor. When using user space, F means that the floating point unit is enabled and V is used to enable the vector unit. On supervisor, it is assumed that floating point and vector units are enabled.

Execution begins when the *RVTEST_CODE_BEGIN* macro is reached and continues until it reaches the *RVTEST_PASS* or *RVTEST_CODE_END* macros, if the test succeeds, or *RVTEST_FAIL* if the test fails. The end macros signal the appropriate execution code to the *FESVR* through the *tohost* register to end the execution. The least significant bit of the *tohost* register indicates whether the message sent to the *FESVR* is a system call that needs to be executed if the value is 0, or program termination if the value is 1.

A test data section should be contained in each test, this section is contained within the *RVTEST_DATA_BEGIN* and *RVTEST_DATA_END*. There is no guaranteed alignment for the start and end of the data section, so regular assembly alignment instructions can be used to ensure desirable alignment of data values. The region of memory which holds the test data section, will be captured at the end of each test to act as a signature from the test.

All test programs contain self-checking code to check the result of the instruction tested. However, this cannot be the only testing strategy since it relies on the correct functioning of the processor instructions used to implement the check.

A timeout facility should be included in any given target environment for running the tests. Tests that do not complete their execution within a given time threshold are classed as failing.

2.2.2 Dhrystone

Dhrystone [15], is a synthetic computing benchmark program created by Reinhold P. Weicker in 1984. The benchmark was first published in Ada in 1984 and later a C version was published by Rick Richardson. The C version of Dhrystone is the most popular one used in industry to measure CPU performance. Dhrystone only measures the performance of integer operations, thus it was named as an integer counterpart of the then popular benchmark *Whetstone*, which was used for floating point operations.

To develop Dhrystone, Weicker gathered information from a wide range of then-popular software and characterized them in terms of various common constructs. It is designed to statistically mimic the processor usage of common sets of programs. Dhrystone's code consists of simple integer arithmetic and logic operations, string operations and memory accesses in order to reflect CPU activity in the most general-purpose computing applications.

Dhrystone produces a *Dhrystones per Second* result, which indicates how many times the program's main loop was executed within a second. A common representation of the program is *DMIPS*, which is obtained by dividing the Dhrystone score by 1757. The number 1757 is the score produced on *VAX 11/780*, which is nominally a 1 MIPS machine. To enable processor performance comparison at different clock speeds, the *DMIPS* measure can be further normalized into *DMIPS/MHz* by dividing the *DMIPS* value by the CPU's clock speed.

Dhrystone possesses features that have allowed it to gain popularity. These features are The portability of its code, allowing it to be easily ported to a large number of platforms and applications, small and easy-to-use and understand code, single easy-to-report score and a free license. Furthermore, its integer code can make it useful as an 8 and 16-bit micro-controller benchmark.

However, Dhrystone has a lot of notable weaknesses [16], [17]. Its' small code can fit into a modern processor's L1 cache memory, minimizing or eliminating the stress on the memory system. Furthermore, compiler optimizations can greatly affect its performance with optimizations such as inline code and dead code removal. A large portion of the benchmark consists of string operations using standard C library functions such as *strcpy()* and *strcmp()* which are highly optimized in common compilers. As a result, Dhrystone's performance and results can vary depending on the compiler used. To address some of these issues an improved version of Dhrystrone [18] was released by Weicker and Richardson in 1988 (Version 2.1). Although this version improves upon compiler optimization dependencies of the benchmark, it does not eliminate them completely. Version 2.1 is the latest version of the benchmark released to date.

In addition, due to its focus on integer operations, it cannot test and demonstrate the capabilities of modern CPUs. It confines itself in the micro-architectural level and does not take into account features commonly used by modern systems such as RISC architectures, sophisticated floating point and vector units, super-scalar RISC designs, VLIW processors, real-time operating systems with sophisticated APIs and graphical user interfaces, failing to represent real-life applications.

Despite its shortcomings, Dhrystone remains quite popular to this day, however, it should not be the only benchmark used when measuring the performance of a system.

2.3 The gem5 Simulator

The gem5 Simulator [19, 20] is a modular discrete event-driven computer system simulator platform, created by merging the best aspects of *M5* [21] and *GEMS* [22] simulators. It offers a flexible, modular and highly configurable design, allowing it to simulate a wide range of systems, that cover a wide range of speed/accuracy trade-offs. gem5 provides a variety of ISA-independent components, such as CPUs, memory models and peripheral devices, and two execution modes with the capability to simulate a full operating system. gem5 supports most commercial ISAs, including ARM, x86, MIPS, ALPHA, Power, SPARC and RISC-V.

2.3.1 Key Features and Capabilities

gem5 is mostly written in *C++* and provides a *Python* interface. All major simulation objects are *SimObjects*, including models of hardware components, such as processor core, caches, buses and peripheral devices, as well as more abstract entities such as workload. Every SimObject is represented by two classes, one in C++ and one in Python. The C++ class defines the SimObject's implementation, state and behavior, while the Python class is used for instantiation, naming and setting the object's parameter values. The simulated system is built using a Python configuration script which provides initialization, configuration and simulation control. All simulated components are defined and configured in the Python script. The standard *main()* function, along with all command-line processing and start-up code is written in Python. When the simulation begins, the Python script is executed almost immediately in start-up.

gem5 supports two simulation modes, System-call Emulation (SE) and Full-System (FS) mode. In SE mode, gem5 executes user-level binaries without executing the kernel-mode system calls of a real operating system. When

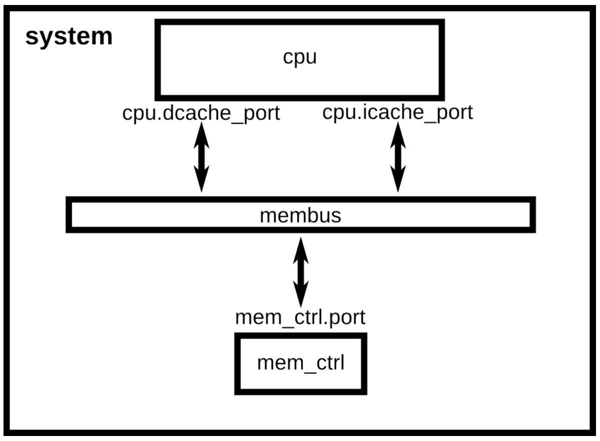


FIGURE 2.4: Simplest system that can be simulated in SE mode, adapted from gem5.org.

the program executes a system call, gem5 traps it and emulates it by passing it to the host operating system. Only Linux system calls are supported in SE mode, and as a result, supports only user-level code. SE mode requires a Python configuration script and a workload binary to run. The simplest system that can be simulated in SE mode is a CPU and a memory system connected with a memory bus, as shown in Figure 2.4. SE mode provides higher simulation speeds and is much simpler to configure since it doesn't simulate all the devices in a system and focuses on the CPU and memory system, however, it is not as accurate as FS mode.

Full-System mode simulates an entire system that is capable of running an unmodified operating system. Along with the CPU and the memory system, FS mode simulates interrupts, privilege levels, I/O devices and exceptions. Full-System mode requires a Python configuration script, an operating system kernel binary, a disk image that contains the file system and a workload binary. The workload binary has to be mounted in the disk image in order to be able to be executed in the simulation. To communicate with the operating system, gem5 provides a serial terminal, called *m5term*, that connects to the system via the system's UART. The FS mode provides higher accuracy compared to SE mode, however, the simulation is much slower due to the complexity of the system simulated. Configuration scripts for FS mode can be more difficult to implement compared to the SE mode since they require the definition and configuration of additional components and peripheral devices. However, gem5 provides a variety of example configuration scripts, for both SE and FS mode, that are highly configurable. Full-System simulation is not supported through all ISAs, due to the complexity of its implementation.

Both SE and FS modes can simulate single-core and multi-core configurations. Furthermore, gem5 supports the simulation of multiple systems by creating another set of objects. These systems can be connected by the user using gem5's network interfaces.

When the simulation finishes, gem5 generates files containing useful information about the simulation along with a diagram of the simulated system. File *stats.txt* includes all measured statistics from the simulation. Files *config.ini* and *config.json* contain all components and parameter values of the simulated system.

Furthermore, gem5 offers advanced simulation features such as fast-forwarding and a checkpoint system. Executed code can be sped up using a simpler CPU model until it reaches a checkpoint, where the execution can continue with a more detailed model. The simulation can be restored at any checkpoint without simulating the previous code. Users can speed up their simulation by inserting checkpoints to important parts of their simulation and restoring them with a more detailed CPU model. Last but not least, gem5 provides debug flags for tracing/debugging of the simulation. Debug flags can be defined at the command line.

2.3.2 CPU and Memory models

There are four CPU models supported by gem5: *AtomicSimpleCPU*, *TimingSimpleCPU*, *MinorCPU* and *O3CPU*.

- **AtomicSimpleCPU:** is a non-pipelined CPU that attempts to fetch, decode, execute and commit an instruction within a single cycle. It is a minimal single IPC CPU that completes all memory accesses immediately. It provides high-speed simulations at the expense of accuracy and can be used for simulation fast-forwarding.
- **TimingSimpleCPU:** is a variation of the Atomic CPU that uses the timing memory accesses. It only allows one outstanding memory request at a time and models the timing of the memory accesses.
- **MinorCPU:** is the detailed pipelined in-order CPU model. Its' standard pipeline consists of two fetch stages, a decode stage and an execute

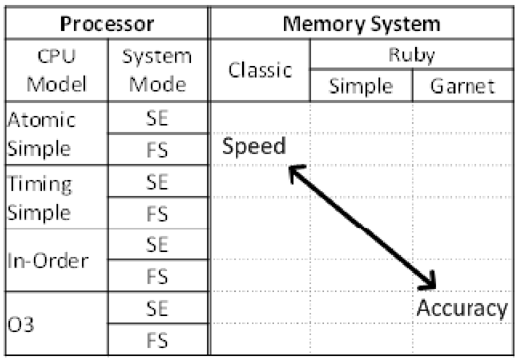
stage which also writes back retired instructions, and it can be configured to simulate CPUs with more stages. The MinorCPU is highly configurable and it is intended to be used to model processors with strict in-order behavior.

- **O3CPU:** is the detailed pipelined out-of-order CPU model loosely based on the Alpha 21264. Its pipeline consists of fetch, decode, rename, issue, execute, write-back and commit stages. It is highly configurable and can model a wide variety of out-of-order CPUs with different pipeline stages and components such as functional units, load/store queues and re-order buffer.

gem5 offers a kernel-based virtual machine (KVM) CPU model that bypasses the simulation and allows binaries running in gem5 to run locally in the host machine if the host ISA is the same as the simulated one in gem5. This model is based on the KVM API in Linux and uses hardware virtualization support available in many modern processors. The KVM CPU can execute at native speed, however, it does not model the timing of execution or memory requests.

The detailed CPU models offer much higher accuracy than the simple CPUs, however, the simulation is slower. Both detailed models follow an *execute-in-execute* model, meaning that an instruction is executed in the execute stage after all dependencies are resolved. Furthermore, visualization of the instruction's position in the pipeline is supported in both detailed models through the *O3 Pipeline Viewer* and the *Minor Viewer* respectively.

gem5 offers two different memory system modes, the *Classic* (adapted from M5) and *Ruby* (adapted from GEMS). The Classic memory system provides a fast, easy and flexible configurable memory system that uses a bus or a crossbar to connect the caches and memories to the system. The Ruby system provides a flexible framework for accurately simulating a large variety of cache-coherent memory systems. The Ruby system offers two network models, *Simple* and *Garnet*. The Simple network models link and router latency as well as link bandwidth, without modeling router resource contention and control flow. The Garnet network models the router micro-architecture in detail, including all relevant router resource contention and flow control timing. To implement a wide variety of cache coherence protocols, gem5 provides a domain-specific language(DSL), named *SLICC* (inherited from GEMS).



Processor		Memory System		
CPU Model	System Mode	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE			
	FS	Speed		
Timing Simple	SE			
	FS			
In-Order	SE			
	FS			
O3	SE			Accuracy
	FS			

FIGURE 2.5: Speed vs. Accuracy trade-offs, adapted from [19]

Figure 2.5 shows speed vs. accuracy trade-offs between gem5's models and execution modes.

2.3.3 ISA Independence

All gem5 CPU models are ISA agnostic to enable the use of each model with every supported ISA. To achieve ISA independence, all CPU models use a common C++ base class to describe instructions. To specify instruction sets, gem5 uses a domain-specific language (DSL), inherited from M5, which unifies the decoding of of binary instructions and the specification of their semantics.

An ISA description file is divided into two sections, the declaration section and the decode section. The declaration section defines the global information required to support the decoder, while the decode section specifies the structure of the decoder and the instructions returned. The decode section includes a set of nested decode blocks that specify a field of a machine instruction to decode and the result to be provided for particular values of that field. Decode blocks are similar to a C switch statement both in syntax and semantics. Each decode block generates a switch statement in the resulting decode function.

In compilation, the description file is parsed and generates C++ code, depending on the instruction format, that implements the behavior of each defined instruction. The generated code overrides functions of the base classes, such as the *execute()* function, to implement each instruction.

2.3.4 RISC-V ISA support

With the growing popularity of the RISC-V ISA, a main feature of gem5's version 20.0 [20], is the added support of the ISA. gem5 supports single and multi-core configurations both on SE and FS mode for the RISC-V ISA.

The base instruction set and extensions were ported from the existing *MIPS* and *Alpha* implementations in gem5, beginning from the 32-bit integer instruction set, RV32I, and then porting the 64-bit RV64 version along with the M extension for multiply/divide instruction support.

The A extension for atomic memory operations includes Load-Reserved/Store-Conditional (LR/SC) instructions and read-modify-write for complex and simple atomic memory operations, respectively. These instructions were implemented as micro-ops that acted like an LR/SC pair. Furthermore, support for single and double-precision floating-point operations was added, implementing the F and D extensions respectively. However, the *round-away-from-zero* rounding mode was not implemented because it is not supported by C++.

The C non-standard extension for 16-bit compressed instructions was implemented because it is used by many RISC-V software toolchains, such as *GCC*. They implemented the extension by adding a state machine in the instruction decoder to detect whether the instruction is compressed or not.

Full-System simulation is supported for RISC-V. Support for SV39 paging scheme with a 39-bit virtual address, a *Page Table Wacker* (PTW), based on the x86 implementation, and *Translation Lookaside Buffers* (TLB) was added.

2.4 Power and Area simulation with McPAT

McPAT [23] is an integrated power, area and timing simulation framework. It supports multi-core and many-core configurations and CMOS technologies ranging from 90nm to 22nm. It is designed to work with a variety of architectural and performance simulators and allows the user to define low-level configuration details and also provides default values when the user specifies only high-level details.

To be able to work with other simulators, McPAT offers an XML-based interface. Users can specify micro-architectural design parameters, such as the number of cores, pipeline width, renaming scheme, cache hierarchy, etc. and

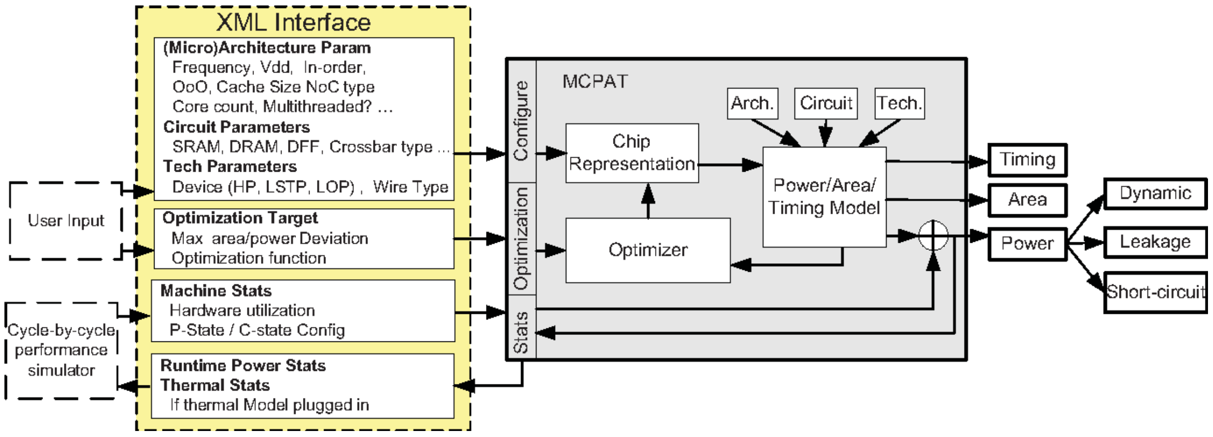


FIGURE 2.6: Block diagram of the McPAT framework, adapted from [23]

performance statistics provided by a performance simulator. The interface allows the definition of circuit implementation and technology details. The XML interface allows McPAT to work independently and only read needed statistics and parameters from other simulators, such as gem5. Furthermore, McPAT output can be used by other simulators, such as *ArchFP* for prototyping pre-RTL floorplans of *VoltSpot* and *HotSpot* for voltage and thermal simulations respectively.

The key components of McPAT are Hierarchical power, area and timing models, an optimizer for determining circuit-level implementations and the internal chip representation that drives the analysis of power, area and timing. Parameters for the internal chip representation are given as input by the user. Figure 2.6 shows a diagram of McPAT's framework, including the workflow of key components.

McPAT has a hierarchical structure that allows low-level architectural modeling, using only high-level details defined by the user. High-level details are used by McPAT's optimizer to determine the low-level details for the internal chip representation, focusing on two major structures, interconnects and arrays. The optimizer produces the final chip representation, on which power, area and timing are calculated.

McPAT's workflow has two main phases, the initialization phase and the computation phase. In the initialization phase, the final chip representation is created using the specified input parameters. For each component, the optimizer performs local greedy optimizations on circuit structures in order to satisfy the timing constraints set by the user. If the resulting area or power does not satisfy the constraints the design is discarded. Configurations that satisfy the constraints are used to create the final design. This chip model is later used to compute the final area, timing and peak power results. The peak power of individual components along with the corresponding utilization statistics is then used to calculate the final runtime power dissipation.

2.4.1 Power, Area and Timing Models

McPAT models dynamic, short circuit and leakage power. Dynamic power is dissipated when circuits charge and discharge capacitive loads to change stages. It is proportional to the total load capacitance, the supply voltage, the voltage swing and the activity factor. The activity factor is determined by the input statistics along with circuit properties, while the load capacitance is calculated by using analytic models for each circuit block of a module. Short-circuit power is calculated using analytical equations. McPAT calculates both subthreshold and gate leakage by determining the leakage current using data from the ITRS and Intel.

To model basic logic gates and regular structures, such as regular logic, memory arrays and interconnects, McPAT uses analytical methodologies from CACTI. For complex structures with custom layouts, McPAT uses an empirical modeling approach that uses published information from existing designs and scales them depending on the target technology.

The timing model uses resistance and capacitance to compute RC delays with methodologies similar to CACTI. Using the delays of components on the critical path McPAT can determine the achievable clock frequency.

2.4.2 Accuracy Errors and Calibration

Despite being quite influential, McPAT lacks support for advanced technology nodes and has known inaccuracies as reported in [24]. Specifically, accuracy error can be high due to:

- **Model Abstraction errors**, meaning that a model for a structure is incomplete or missing, or the implementation of the model is too high-level and misses low-level details.
- **Modeling Assumption errors**, when the implementation of a micro-architectural structure differs from that of McPAT.
- **Input Errors**, arising from unknown or incorrectly specified input parameters.

- **Coding Errors**, generated from programming mistakes or bugs in the tool. However, most coding errors have been resolved in the latest version of McPAT.

To reduce the impact of said errors, there are many available frameworks that calibrate McPAT's output results using machine learning models, such as *PowerTrain* [25] and McPAT-Calib [26]. To extend on the supported technology nodes, McPAT-PVT [27] and McPAT-7nm [28], add support for up to 22nm and 7nm FinFET technology nodes respectively. Furthermore, McPAT-Monolithic [29] adds support for 3-D Hybrid Monolithic multi-core architectures.

Chapter 3

Related Work

3.1 gem5 Simulations of ARM and x86 Processors

The gem5 simulator is an open-source computer architecture simulator system. gem5 offers support for a wide range of instruction set architectures and highly configurable models along with a straightforward front-end interface. This, combined with fast simulation times and cycle-level accurate simulations, makes gem5 a popular option for architectural and micro-architectural system simulation and exploration. This section showcases notable works regarding the accuracy and use of the gem5 simulator.

In [30], Butko et. al. simulated a *Cortex-A9* CPU using gem5, focusing on the accuracy of execution time on selected benchmarks. They simulated a dual-core ARM *Cortex-A9* running at 1 GHz, with private L1 data and instruction caches and a shared L2 cache, based on and validated against the configuration of *Snowball SKY-S9500-ULP-C01* hardware development kit. The authors run various benchmarks from *SPLASH-2*, *ALPBench* and *STREAM* suites on gem5's Full-System mode and on the *Snowball SDK*, disabling its' *DVFS* feature to ensure that the system runs at a constant 1GHz. They report a mismatch in execution time, meaning the time it takes the simulated system to finish execution, between 1.39% and 17.94% between the real device and the simulated system. Furthermore, to demonstrate gem5's architectural exploration capability, they modeled hypothetical multi-core systems made of up to 8 cores, resulting in speedups ranging from 3.51 to 7.03.

In [31] Endo et. al. focused on in-order and out-of-order simulation, by matching the performance of two ARM CPU cores, *Cortex-A8* and *Cortex-A9*, using gem5's CPU models. At the time this work was published, gem5 did not provide a functional in-order CPU model for ARM. As a result, the authors use gem5's *O3CPU* model for both cores, modifying it to simulate an in-order pipeline in the case of *Cortex-A8*. The parameters used to configure their CPU models were based on the development kits *Snowball SKY-S9500-ULP-CXX* for the *Cortex-A9* core and *BeagleBoard-xM* for the *Cortex-A8* core.

To evaluate the accuracy of their models, they simulated a number of benchmarks from the *PARSEC 3.0* suite on a single-core configuration with a L2 cache for the in-order model and a dual-core configuration with a shared L2 cache, both on a Full-System configuration on gem5. Comparing the simulation results with the results produced by their reference models, *Snowball SDK* for the out-of-order model and with the *BeagleBoard-xM SDK* for the in-order model, they achieved an execution time absolute error of 7.4%, ranging from 1 to 17%, for the *Cortex-A9* model and 8%, ranging from 2 to 16%, for the *Cortex-A8* model. To showcase the design space exploration capabilities of gem5, they simulated a dual *Cortex-A8* configuration with a shared L2 cache, resulting in an almost perfect average speedup of 1.77 over its single-core version.

Expanding on their work, Endo et. al. in [32] simulated CPUs of a *big.LITTLE* system consisting of ARM *Cortex-A* cores, and presented performance, power and area metrics using gem5 and McPAT. They simulated a *big.LITTLE* system with two clusters of four cores each, one with *Cortex-A7* and the other with *Cortex-A15*, configuring the models of their previous work described in [31]. The configuration of the simulated system was based on the *ODROID-XU3* board which embeds an *Exynos 5244* Application Processor.

Running the *Dhrystone* benchmark and a number of benchmarks from the *PARSEC 3.0* suite, they achieved a 3.6% error on the area estimations for the A15 and exactly matched the area of the A7. In the cluster estimations, they had a -13 and -1.4% mismatch for the A7 and A15 clusters respectively. For energy and performance, they simulated only one active core for each cluster running single-threaded benchmarks. In the *Dhrystone* benchmark they achieved a speedup of 1.84 for the A15 and 3.69 times less energy consumption for the A7 core, which is very close to the results published by ARM. For the *PARSEC* benchmarks they achieved, on average, a speedup of 1.5 for the A15, which differs from the reported speedups of 2-3x, and 4.1 times less energy consumption for the A7.

Butko et. al., in [33], also focused on the evaluation of performance and power models of the ARM *big.LITTLE* architecture implemented in gem5 and McPAT simulation frameworks. They simulated a system with two CPU clusters of four cores each with private L1 data and instruction caches and shared L2 cache in each cluster, based on the *Exynos 5 Octa (5422)* SoC, which consists of *Cortex-A7* and *Cortex-A15* cores. For their implementation, they used gem5's in-order and out-of-order models, fine-tuning their parameters, to match the specifications of the *Cortex-A7* and the *Cortex-A15* cores respectively. In contrast to the work reported in [32], the authors in [33] focus on the multi-core evaluation of the *big.LITTLE* system, performing modifications on gem5 and the Linux kernel to be able to simulate both CPU clusters and boot all 8 cores simultaneously in FS mode.

Their models were validated against the *ODROID-XU3* board which is built around the *Exynos 5 Octa (5422)*, on various benchmarks from the *Rodinia* and *Lmbench* suites. They report that the tests run on three different sets of static frequency on both clusters, with the *DVFS* features disabled, both on gem5 and real hardware. They report an average absolute error on performance of 18.8% for the *LITTLE* cluster, 20.1% for the *big* cluster and 22.9% for the *big.LITTLE* system. On the total power consumption, they report a mismatch of 12.7%, 11.7% and 10.8% for the *LITTLE* cluster, *big* cluster and *big.LITTLE* system respectively. Last but not least, the reported average absolute error for energy-to-solution is 21.9%, 27.9% and 22.1% for the *LITTLE* cluster, *big* cluster and *big.LITTLE* system respectively.

To evaluate the performance of ARM and Intel x86 architectures in gem5, Abudaqa et. al. conducted a comparative study in [34], simulating benchmarks on multiple configurations on both architectures. The authors simulated two in-order and two out-of-order single-core configurations with L1 data and instruction caches and a L2 cache on both architectures using gem5 in-order and out-of-order CPU models respectively. They simulated a number of benchmarks from the *Mibench* suite on SE mode, comparing the two architectures on average CPI, L2 cache miss rate, total energy and throughput. They report that ARM outperforms x86 in most cases, however, the difference between architectures is minimal when the CPU model is out-of-order. Regarding the L2 cache miss rate metrics, they report that the miss rate becomes higher as the size of the caches is doubled, which is not reasonable. They justify this result as their simulation focuses on gem5 SE mode, which produces less accurate results.

3.2 gem5 applications and implementation on the RISC-V ISA

RISC-V is an open-source instruction set architecture, which has gained popularity from both academia and the industry by improving upon mistakes of other open-source RISC architectures [3]. Its accessibility, due to being open-source, makes RISC-V a great addition to the gem5 simulator since many of the instruction sets already supported are proprietary and require licenses that can be costly and difficult to work with. In this section, we summarise important works regarding the use of gem5 and its' accuracy in the context of RISC-V.

In [35], Roelke and Stan introduce *RISC5*, an implementation of RISC-V for gem5, which supports the standard instruction set with the common extensions as well as the compressed instruction set for single-core simulation in system call emulation (SE) mode. To implement the instruction set, the authors adapted most the code gem5 uses to implement MIPS, implemented atomic operations and floating point instructions. They report that instruction *eret* is not implemented since SE mode does not support privilege levels. To showcase gem5's compatibility with external tools, they create a tool flow where they execute two RISC-V cores, Rocket and BOOM. Simulating one million instructions of the *libquantum* benchmark, the authors report that Rocket is much larger than BOOM due to its significantly larger L2 cache, however, BOOM consumes more power and achieves significantly higher temperatures due to being out-of-order and having higher associativity.

To validate their implementation, they simulated a number of benchmarks from the *SPEC CPU2006* suite on gem5, a Chisel-generated C++ simulator and on a RISC-V softcore on an FPGA. They report that gem5 is accurate in the number of retired instructions, number of memory operations and number of executed branch instructions, and less accurate in the total number of cycles for each benchmark and the number of instructions fetched. Furthermore, it is reported that the FPGA took about 26.5 times less time than gem5, on average, to execute the benchmarks, while the Chisel simulator took, on average, 32 times more. Notice that the authors, refer to simulation time, meaning the time it took the simulator to finish execution, comparing the performance of the simulators.

To expand on the previously mentioned work, Ta et al. in [36] implement support for multi-core simulation of RISC-V systems in gem5 for SE mode.

To implement multi-core support, the authors modified gem5 to support thread-related system calls and RISC-V synchronization instructions. Since the implementation of system calls in SE mode is mostly ISA-independent, hence, already implemented in gem5 for RISC-V by default, they report that the only system calls implemented are: *clone*, which spawns a new thread, *futex*, used for OS-level thread synchronization, and *exit*, which terminates a thread's execution. Furthermore, they implemented atomic memory operations and load-reserved/store conditional instructions.

To validate their implementation, they used a number of assembly and low-level C benchmarks from the *RISC-V tests* suite to test the single-threaded implementation and for the multi-threaded implementation, they developed a number of assembly unit tests. They developed an assembly micro-benchmark that tests the multiplier's performance in gem5's in-order CPU and validated it against the multiplier performance of Rocket Chip. To evaluate the performance of their implementation, they run a number of benchmarks from the *Ligra* suite on a configuration of Rocket Chip in gem5 and compare the results against a Chisel C++ RTL simulator. They report that the RTL simulator simulated slightly more instructions than gem5 and it was more than an order of magnitude slower compared to gem5. Finally, they run several benchmarks using the *OpenMP* run-time on systems with different numbers of in-order cores. They report that gem5's performance scales well with the number of simulated CPU cores.

The next step to fully support RISC-V in gem5 is implementing Full System simulation. Yuen, Liao et. al. present a Full System implementation for RISC-V in gem5 in [37]. They added a platform class, called *HiFive*, which is based on *SiFive's HiFive* series of boards with memory map conventions and peripheral addresses based on the *SiFive U54MC SoC* dataset. Furthermore, they added a *Core Level Interrupt Controller (CLINT)* component, which handles software and timer interrupts, and a *Platform Level Interrupt Controller (PLIC)*, that handles the routing for external interrupts, as well as a *Physical Memory Attribute (PMA)* checking mechanism for checking attributes of memory addresses. They report that the *VirtIOMMIO* is ported from the *ARM* configuration while the *UART* module is already built-in in gem5. To complete the implementation, they also added a number of fixes in *Privileged Instructions* and interrupt handling logic as well as a *Device Tree Blob (DTB)* file generation feature. Finally, they report that the checkpointing functionality is verified and working on all CPU models of gem5.

The authors ran several experiments to verify their implementation. They simulated a system with four CPU cores using the Berkeley Bootloader with Linux kernel v5.10 along with the file system of the *BusyBox* disk image. They report that the system was able to successfully boot up Linux and execute commands using the terminal. Furthermore, they run a number of benchmarks from the *PARSEC* suite, with 1, 2 and 4 threads on a configuration with four out-of-order CPU cores and *1024M* of DRAM. They report 39ms execution time for the *Blackholes* benchmark using 1 thread and 10.1ms using 4 threads, resulting in a speed-up that is close to 4. To conclude, they simulated a system with one CPU core and one hardware thread, where they run Linux as a guest OS of *Diosix Hypervisor*. They report that the simulated run-time is increased as they increase the number of threads used to run the benchmark.

In [38], Chatzopoulos et. al. used gem5's SE mode to simulate a model of the RSD RISC-V processor and compared it to an RTL simulation baseline, in order to point out the accuracy of the simulators, the challenges and possible sources of error. The authors simulated an out-of-order configuration of the RSD processor, using gem5's *O3CPU* model with a L1 instruction and data cache of 4kB each and a *gshare* branch predictor, which they implemented from scratch. Since the RSD model is built upon the RV32I base instruction set and gem5 runs the RV64I instruction set, they used only programs that use 32-bit sizes for both the RSD model and gem5's SE mode, compiled with the RISC-V GCC cross-compiler without optimizations.

The authors used a set of custom benchmarks along with three benchmarks from the *MiBench* suite, that tackle specific parts of the processor to evaluate the simulation accuracy of both models and to retrieve important microarchitectural parameters of the reference model. Using gem5's detailed out-of-order model, they report a simulation time speedup of 5x up to 20x compared to the fast behavioral simulation of the RTL model. To measure runtime accuracy, the authors used a pipeline visualizer and compared the instruction pipeline of both models. They report that gem5 reports 36% more clock cycles in the *BubblesortC* benchmark and 35% less in the *StringSearchSmall* benchmark, compared to the RTL model, while the rest of the benchmarks show little differences between the two models. Furthermore, they report that the number of committed instructions of both models is virtually the same in most cases. The authors conclude that the memory system and the branch predictor are the most significant factors that affect the accuracy of the micro-architectural model.

3.3 Power modeling using McPAT

Power is an essential factor when designing architectural systems, especially on embedded and mobile systems. As a result, there is a great demand for architectural power modeling tools that offer fast simulations and accurate models to estimate power consumption on a wide range of systems. McPAT is a power modeling tool that offers power, area and timing estimations and combined with a straightforward interface has gained popularity among researchers. Therefore, the accuracy of McPAT's models is of great significance.

In [24], Xi et al. performed the first highly detailed analysis of the cause of potential errors on McPAT's power and area estimations. They created and simulated three models of IBM's *POWER7* CPU in McPAT, using performance statistics generated from an IBM performance simulator for *POWER* chips. To evaluate their models, they used several benchmarks from the *SPEC2000* and *SPEC2006* suite, comparing the results between McPAT and an IBM power modeling tool. They report that the main sources of error in McPAT occur from abstraction errors in McPAT's models, modeling assumption errors and input errors.

To improve upon McPAT's accuracy, Lee et al. [25] introduce *PowerTrain*, a learning-based calibration framework on McPAT's models. To validate their framework, they use the *Cortex-A15* as their baseline processor and the *ODROID-XU3* board with only one *Cortex-A15* core enabled for power measurement. They executed benchmarks from *MiBench* and *SD-VBS* suites on the real hardware and they used the execution statistics as input on four different baseline models for McPAT. After the calibration, they report 2.41% mean percentage error and 4.37% mean percentage absolute error between the calibrated and the measured power.

Zhai et al. [28] introduce *McPAT-7nm*, a modified version of McPAT that supports FinFET technologies up to 7nm, along with *McPAT-Calib*, a framework that uses advanced machine learning methods to calibrate *McPAT-7nm*, to obtain more accurate modeling results. They evaluated their model using 80 benchmarks from various suites along with 15 different configurations of the RISC-V *BOOM* core. The authors used *7nm ASAP7 PDK* and commercial gate-level power analysis flow to obtain the power ground truth. They report a mean absolute percentage error of 4.47% on leakage power using

Poly_SVR, 7.40% on dynamic power using *XGBR* with the total amount of features and 6.23% using *XGBR* with dynamically selected features.

To integrate McPAT along with gem5, Tashiro and Oyamada [39] introduce *VIPEX* (Virtual Platform Exploration), an environment that integrates both simulators and improves their accessibility by incorporating a simple graphical interface. The authors present a case study of an exploration of a design space with 17496 architectures with various single-core and multi-core configurations, based on gem5's detailed ARM CPU model, with a workload of a matrix multiplication. They were able to determine the best and worst architectures regarding area, power, execution time and energy and analyze the trade-offs between configurations. However, their environment uses an exhaustive method, as it is under development, and as a result, it is too slow.

3.4 Thesis Approach and Motivation

In most of the projects mentioned above, the authors focus on coarse-grain performance comparisons, such as bulk execution time, without reporting on the causes of irregularities in their results. In this project, we will compare the performance and energy costs between real and simulated RISC-V systems, focusing on instructions per cycle (IPC), and attempt to define the sources that cause the deviations from the real system. We chose Ariane, as we have available information on its performance and design parameters.

We will compare Ariane's reported performance with configurations of gem5, using gem5's detailed models on Full-System mode. Then, we compare Ariane's reported leakage power and energy with our configurations using McPAT.

Chapter 4

Simulation Environment Setup and Implementation

This chapter presents the implementation for modeling and simulating the target system. It describes tool workflow along with the tool use and setup and the functionality of the models used.

4.1 Concept and Tool Work-Flow

As we mentioned above, the goal of this thesis is to compare the energy costs and the performance between a simulated environment and an actual ASIC implementation of RISC-V cores to provide a preliminary fault analysis of the former. Towards this objective we create a simulation model that matches, as far as possible, the ASIC implementation of CVA6 (former Ariane) [2]. gem5 [19, 20, 40] offers a fast simulation and a large number of highly configurable micro-architectural models. This makes it suitable for providing the micro-architectural simulation and useful event statistics of the target system. For this project, we use version 21.2.1 of gem5, built for the RISC-V ISA. To obtain analytical power and area statistics, we use a modified version of McPAT [23] called cMcPAT [41]. cMcPAT is a modified version of McPAT employed in the COSSIM simulator framework [42] that can also be used independently as a stand-alone package. It incorporates changes integrated into McPAT v1.3 that address the issues reported in [24] improving the accuracy.

In [2], it is reported that a number of assembly-level tests were developed to exercise particular architectural elements and provide classification for different instruction groups and hardware modules commonly found in RISC-V architectures. The authors provide a list of energy and leakage power results produced by these particular tests. Furthermore, the authors report IPC results for the Dhrystone benchmark. For the purposes of this project, we decided to use the RISCV-Tests suite which offers a large number of assembly-level compliance tests and low-level C benchmarks, including Dhrystone.

4.1.1 Tool Work-Flow

We begin the flow by writing a gem5 configuration file that includes all the provided micro-architectural design parameters. These configuration files are part of the simulator's front-end environment that is written in Python which is responsible for instantiating and orchestrating the models in the back-end which are written in C++. After running each benchmark under this configuration, gem5 produces a large number of event and performance statistics. Given gem5's output files along with a template of McPAT's input XML file to cMcPAT's integrated *GEM5toMcPAT.py* script, we are able to produce an input XML file that can be passed to McPAT to produce power and area estimates. Using McPAT's power results alongside the total run-time from gem5, we obtain both energy estimations and performance results for the target system. In Figure 4.1, we present the flow described above.

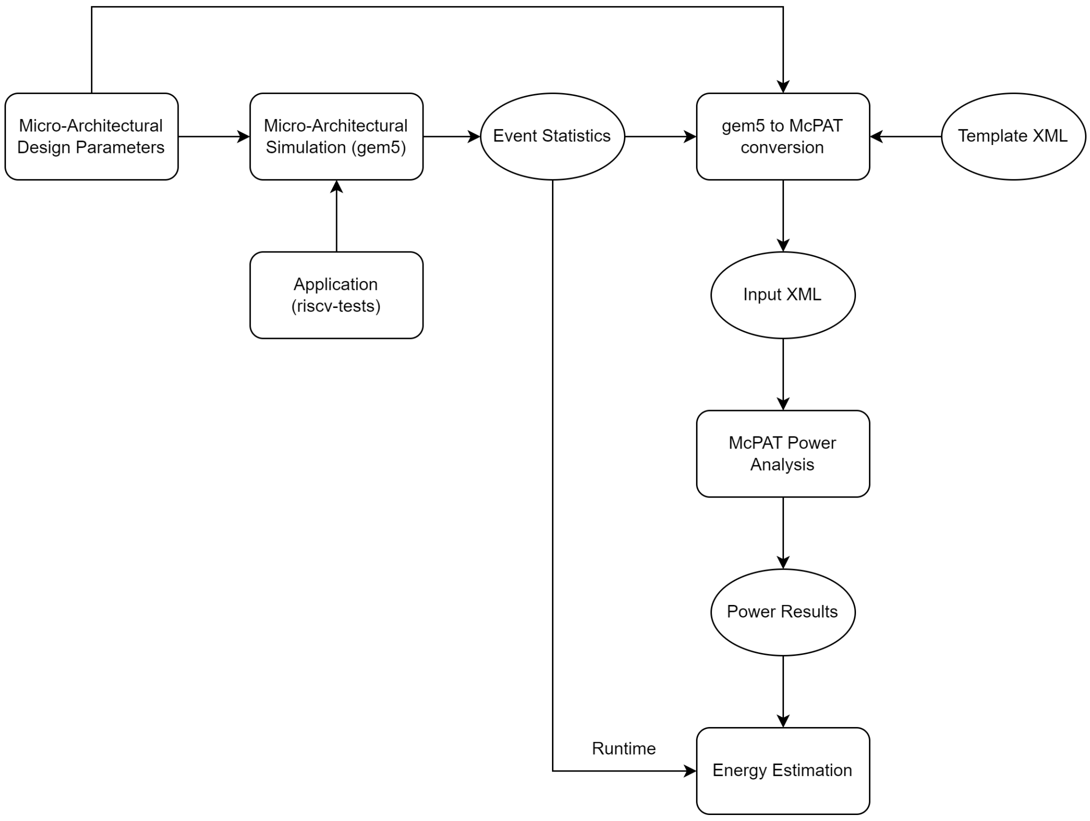


FIGURE 4.1: Tool work-flow.

4.2 Benchmarks

In [2], they report that they developed a number of assembly-level tests that test particular architectural elements. Specifically, they test: *ALU instructions* for arithmetic and logic operations, *Multiplications*, *Divisions*, *Load and Stores* with and without virtual memory and a *Mixed Workload* using a generalized matrix-matrix multiplication. They also report IPC and DMIPS/MHz produced by the *Dhrystone* benchmark.

To be able to measure the performance and required energy of our model and compare it to the results listed in [2], we selected a number of tests and benchmarks from *RISCV-Tests* suite. *RISCV-Tests* provide a number of assembly-level tests and low-level C tests built for bare-metal systems that match the description of the workload used to measure the performance in [2]. In particular, we chose: *rv64ui-p-add* for ALU instructions, *rv64um-p-mul* for multiplications, *rv64um-p-div* for divisions, *rv64ui-p-sw* for Load/Stores with virtual memory disabled and *rv64ui-v-sw* for Load/Stores with virtual memory enabled, from the ISA tests of the suite. For the mixed workload, we chose the matrix multiplication benchmark included in the *riscv-tests* suite. Last but not least, to obtain the IPC of our CPU, we used the Dhrystone v2.2 benchmark included in the *riscv-tests* suite, in our bare-metal configuration. Unfortunately, we could not obtain the Dhrystone score in DMIPS/MHz, as gem5's bare-metal setup does not support the serial terminal. To solve this, we used Dhrystone v2.1 in Full-System simulation. Versions 2.1 and 2.2 of the Dhrystone benchmark are functionally identical.

4.2.1 Setting up the tests

The assembly tests from the suite are designed for ISA compliance testing and functional validation of architectural components. These tests do not produce meaningful performance and energy statistics since their code is short. To solve this, we modified the main section of the tests to run in a loop of 200 iterations using a simple branch loop.

To compile the tests, the *riscv-gnu-toolchain* package, which contains the GNU GCC cross-compiler for RISC-V, has to be installed. After setting up the RISC-V toolchain, before compiling the test suite, the Linux *RISC-V* environmental variable, `$RISCV`, has to be set to the *RISC-V tools* install path and the variable has to be added to the Linux path variable, `$PATH`. The test suite is then built using *riscv64-unknown-elf-gcc* cross-compiler. After compilation, the test and benchmark binaries are generated in the suites directory. More information about setting up the RISC-V toolchain and compiling the *risv-tests* suite is provided in Appendix A.

To set up Dhrystone for the Full-System configuration, we compiled Dhrystone v2.1 with *riscv64-linux-gnu-gcc* for 10 million iterations, with the same compilation flags as in the bare-metal version. The *-O2* optimization flag was enabled in both setups. We tested that Dhrystone exits with a "measured time too small" message and does not produce a score when compiled for less than 10 million iterations.

4.3 Implementing the core model for simulation in gem5

From [2], we obtain the basic architectural design parameters listed in Table 4.1. Notably, we state here that the original study [2] does not describe all the parameters in detail. To sidestep this we set default values for parameters that are omitted by the authors. For transparency, we annotate which of them are known and which of them are set to default. For sanity check, we also measure the impact of the default parameters to the overall analysis, by slightly tweaking their values.

To be able to pass the given design parameters and hardware modules into our simulation model, we have to create a Python configuration file that contains all the necessary classes (SimObjects) for the simulation.

File *ariane.py* was created and contains the following modules:

- Class for configuring CPU
- Classes for configuring instruction and data Cache
- Branch Prediction Unit configuration class
- Classes for ALU, LSU, FPU and Mult/Div configuration
- Functional Unit Pool class

TABLE 4.1: Architectural design parameters reported in [2].

Parameter	**Chosen**
BHT	8
BTB	8
ROB Entries	8
Fetch latency	1
L1 I-cache	16kB, 4-way
L1 D-cache	32kB, 8-way
L1 D-cache latency	3
Integer ALU latency	1
Register File	31x64 flip-flops
I-TLB Entries	16
D-TLB Entries	16
Clock	1.7GHz

In subsequent subsections, we analyze the aforementioned configurations in further detail.

4.3.1 CPU Configuration

Ariane has a 6-stage, in-order, single-issue pipeline. Its front-end consists of a PC generation stage and an instruction fetch stage. Instructions are issued and committed in-order, however, they can retire out-of-order. For that reason, we decided to use gem5's out-of-order CPU model, *DerivO3CPU*. gem5's in-order CPU model, *MinorCPU*, does not contain modules that affect Ariane's performance, such as the Reorder Buffer, and cannot execute instructions out-of-order. Thus, we assume that MinorCPU is not suitable for our case.

We create class *ARIANE()*, which inherits from gem5's *O3CPU* class. This allows us to configure gem5's *DerivO3CPU* with our CPU's micro-architectural design parameters. We are able to configure the pipeline's width and depth, the number and size of Reorder Buffer (ROB), Instruction queue and Load/-Store queue size, branch predictor type and functional units.

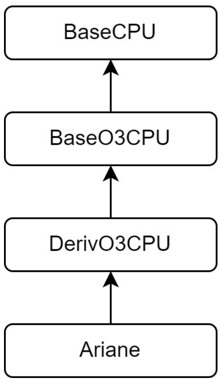


FIGURE 4.2: CPU class inheritance diagram.

4.3.2 Instruction/Data Caches

For Ariane's instruction and data cache, we create classes *ARIANE_ICACHE()* and *ARIANE_DCACHE()*. These classes inherit from gem5's *Cache* class and allow us to configure the size, associativity and latency for each cache memory. Although the *Ruby* model is more accurate, we chose the classic *Cache* model, since the Ruby model is used mainly for cache coherency protocols and complex cache hierarchies. In our case, we only have L1 data and instruction cache, thus the Ruby model is unnecessary.

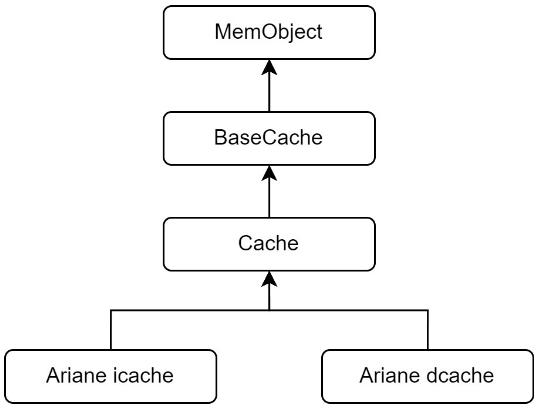


FIGURE 4.3: Cache class inheritance diagram.

4.3.3 Branch Prediction Unit

Ariane employs a Branch History Table, Branch Target Buffer and a Return Address Stack. To configure the branch prediction unit, class *ARIANE_BR()* was created and it inherits from gem5's *LocalBP* class, which allows us to configure BHT, BTB and RAS parameters.

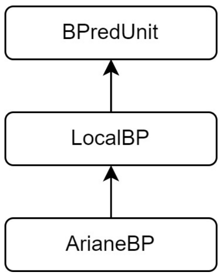


FIGURE 4.4: Branch Prediction class inheritance diagram.

4.3.4 Functional Units

In [2] it is reported that Ariane has a total of 6 functional units:

- ALU: handles most of the integer base RISC-V ISA
- LSU: manages integer and floating-point load/stores
- FPU: handles floating-point operations.
- Multiplier/Divider
- Branch unit: handles branch prediction and branch correction, and
- CSR: handles CSR operations

To configure the functional units, gem5 offers the *FUDesc* class. This class has 2 fields, *Count* and *OpList*. Parameter count defines the number of functional units of the same type and *OpList* is a list of *OpDesc* objects. *OpDesc* defines the type of operations a functional unit handles via the *OpClass* parameter and the latency of this type of operations by *OpLat* parameter.

To model and to be able to configure the ALU, LSU, FPU and Mult/Div, classes *Ariane_ALU*, *Ariane_LSU*, *Ariane_FPU* and *Ariane_MultDiv* were created respectively.

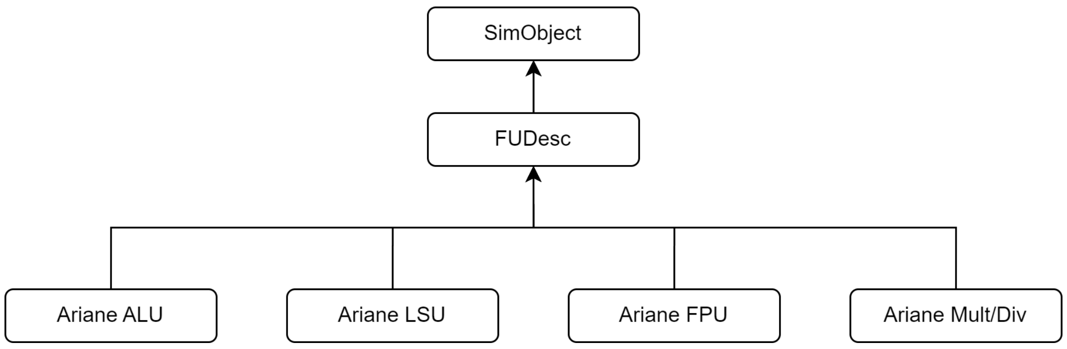


FIGURE 4.5: Functional Units class inheritance diagram.

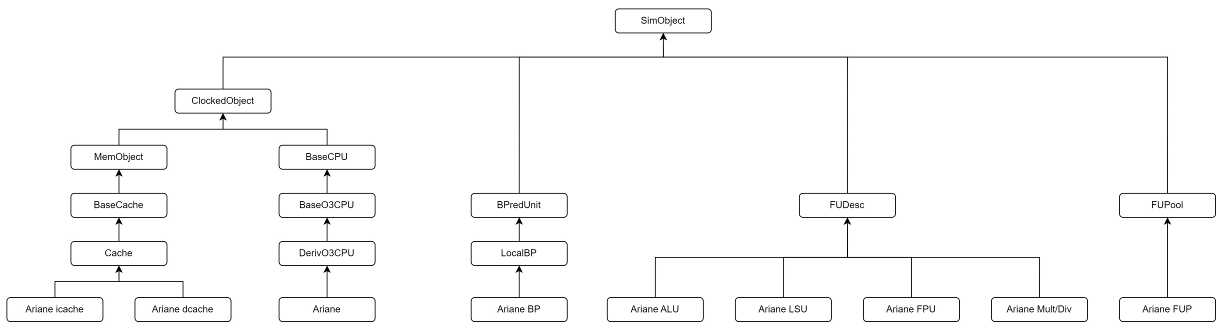


FIGURE 4.6: Class inheritance diagram of objects in ariane.py.

gem5 does not have a dedicated *OpClass* that handles branch prediction. It offers an *OpClass* named *SimdPredAlu* that handles SIMD branch prediction, however, since Ariane does not have a SIMD unit we decided not to use it. Furthermore, CSR instructions belong to gem5's *No_OpClass*. As a result, the Branch and CSR unit could not be modeled.

To be able to pass the functional units to the CPU model, gem5, offers class *FUPool*, that stores all functional units needed for the simulation in a list. *Ariane_FUP* class was created and it inherits all the attributes of class *FUPool*. Our FUPool contains the ALU, the Floating-Point Unit, the Load/Store Unit and the Multiplier/Divider, as shown in Figure 4.7.

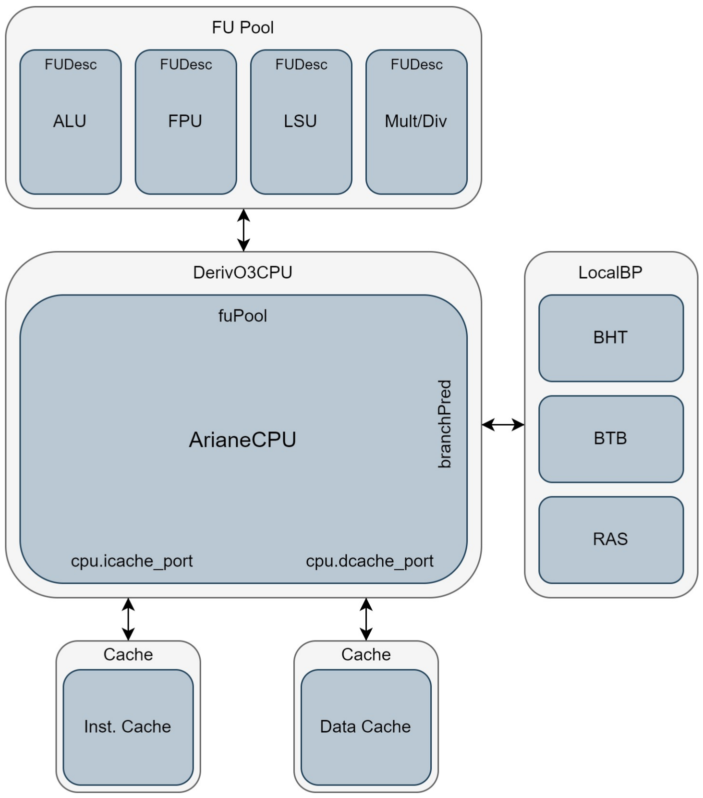


FIGURE 4.7: CPU configuration for simulation in gem5.

Figure 4.7 shows a diagram of the CPU with the modules created in the file *ariane.py*. The blue blocks are the new objects created in *ariane.py* while the gray blocks are their parent classes. The instruction and data caches are connected to the CPU's *cpu.icache_port* and *cpu.dcache_port* respectively. The branch prediction unit is set to the parameter *branchPred* and the Functional Unit Pool is set to the parameter *fuPool*. This script is used to initialize the CPU and the caches of our system, by instantiating the respective objects in the Full-System configuration file described in later sections. Buses, peripheral devices and memory are also defined and instantiated in the Full-System configuration file.

4.3.5 Choosing Parameter Values For The Base Model

In this section, we indicate the selected values for the architectural design parameters of the base configuration.

To begin with, in [2], they report that the ALU handles most of the base RISC-V ISA and it has a fixed latency of 1 cycle. For that reason, we chose the *IntAlu* operation class and we left the default latency of 1 cycle.

TABLE 4.2: Chosen Functional Unit parameters.

Functional Unit	**OpClass**	**Latency**	**Pipelined**
ALU	IntAlu	1	-
Mult/Div	IntMult	2	True
	IntDiv	2,16,32,64	False
LSU	MemRead	2	-
	MemWrite	2	-
	FloatMemRead	2	-
	FloatMemWrite	2	-
FPU	FloatAdd	2	-
	FloatCmp	2	-
	FloatCvt	2	-
	FloatMult	4	-
	FloatMultAcc	5	-
	FloatMisc	3	-
	FloatDiv	14	False
	FloatSqrt	24	False

We selected *IntMult* and *IntDiv* operation classes for the Mult/Div unit of our simulated system, since the Multiplier/Divider of Ariane handles integer multiplications and divisions. They report that the multiplier is fully pipelined and has 2 stages, thus, we assign a latency of 2 cycles and enable the *Pipelined* parameter. The divider is not pipelined, and as they report divisions can take from 2 to 64 cycles depending on the operand values. Since we cannot apply a range in the latency parameter, we will simulate our benchmarks for four different cases. With 2 and 64 cycles for the best and worst cases respectively, and with a latency of 16 and 32 cycles for two middle cases.

They report that the Load/Store Unit manages integer and floating point loads and stores. Therefore, we selected the appropriate integer and floating point operation classes. Furthermore, we chose all the corresponding operation classes for floating-point operations for the Floating-Point Unit, as shown in Table 4.2. The authors of [2] do not provide any additional information on the latency of the FPU and the LSU, consequently, we left the default values of gem5, as they seem reasonable for the respective operations. Table 4.2, indicates the selected parameter values of the functional units of our simulated system.

TABLE 4.3: Chosen Branch Prediction Unit parameters.

Parameter	**Config A**	**Config B**
localPredictorSize	8	128
localPredictorBits	2	2
BTBEntries	8	64
RASSize	2	2

Ariane supports branch prediction via a Branch History Table with a 2-bit saturating counter, a Branch Target Buffer and a Return Address Stack. In [2], the authors report that the ASIC implementation of Ariane has a BHT and a BTB of 8 entries each. However, they cite an alternative configuration of a 128-entry BHT and a 64-entry BTB with which they report an IPC of 0.82. As a result, we will run our experiments on both configurations and measure their impact on performance. The selected values of both configurations are shown in Table 4.3. The value of RAS was set at 2 entries based on Ariane's source code.

Based on [2], the instruction cache is 16kB in size with an associativity of 4 and the data cache has a size of 32kB, an associativity of 8 and a latency of 3. We set the corresponding values as shown in Table 4.4. We set the latency of the instruction cache at 1 cycle because they report a fetch latency of 1 cycle. Since Ariane does not support MSHRs, we set their corresponding parameters at the lowest possible value, which is 1.

TABLE 4.4: Chosen Cache parameters.

Parameter	**I-Cache**	**D-Cache**
size	16 kB	32 kB
assoc	4	8
tag_latency	1	3
data_latency	1	3
responce_latency	1	3
mshrs	1	1
tgts_per_mshr	1	1

gem5's out-of-order model has a 7-stage pipeline. However, by changing the delay parameter values, it can simulate pipelines with different numbers of stages. In order to simulate the pipeline of Ariane, the ideal solution would be to assign a delay value of 0 to one of O3CPU's stages, specifically to the *Rename* stage since CVA6 does not have a dedicated renaming stage. Unfortunately, after testing such configurations, the simulation would stall and crash. As e result, we set the delay parameter values to the lowest possible value of 1.

Based on Ariane's source code, we noticed that Ariane fetches 2 instructions in one cycle when the instructions are compressed and 1 if not. Furthermore, in [2], they report a fetch width of 32 bits and that CVA6 fetches 1.5 instructions on average, due to a large percentage of the instructions being compressed. Based on the above, we decided to set a fetch width of 2 instructions in the base model. CVA6 is a single-issue processor, thus the issue width was set to 1. The commit width value was set to 2 as reported in [2]. The rest width parameter values for the base model were assigned as shown in Table 4.5.

TABLE 4.5: Chosen Pipeline Width values.

Parameter	**Value**
fetchWidth	2
decodeWidth	1
renameWidth	1
dispatchWidth	1
issueWidth	1
wbWidth	1
commitWidth	2
squashWidth	4

Since they report a fetch width of 32 bits, we assigned a 4-byte fetch buffer and a fetch queue of 4 entries based on the source code. It is also known from [2] that the Reorder Buffer has 8 entries. However, information on the Load/Store queues, as well as the instruction queue was not provided in [2]. Therefore, judging by the fact that Ariane has a small circuit, we selected 2 entries for the Load/Store queue and 8 entries for the instruction queue. Table 4.6 lists the selected CPU parameter values for the base configuration.

TABLE 4.6: Chosen CPU parameter values.

Parameter	**Value**
fetchBufferSize	4(Bytes)
fetchQueueSize	4(Entries)
LQEntries	2
SQEntries	2
numIQEntries	8
numROBEntries	8
numPhysIntRegs	40, 44
numPhysFloatRegs	40, 44
I-TLB	16
D-TLB	16

We selected 16 entries for instruction and data TLB each, as reported in [2].

4.4 Configuration files setup

Tests from the *riscv-tests* suite are built for bare-metal systems and run in machine mode (M-mode). Systemcall Emulation (SE) mode supports binaries running in user mode (U-mode), thus, running the tests in SE mode returns an *Illegal Instruction* error. As a result, SE mode is not suitable for our case.

To simulate binaries that run in M-mode in gem5 we need a bare-metal setup, which can be accessed through Full-System (FS) simulation. FS mode has the following requirements:

- An FS configuration script.
- A Linux Kernel Binary.
- A disk image containing the file system.

The configuration script instantiates and initializes our system and runs the simulation. By using *RiscvBareMetal()* as the system's workload and giving the test binary in the *bootloader* parameter of workload, the simulation runs on bare-metal mode. Unfortunately, the bare-metal setup does not support the serial terminal and as a result, we cannot obtain the score produced by Dhrystone. For this reason, we have to configure our script to be able to run both the bare-metal setup and regular Full-System simulation, by letting the user choose between the *RiscvBareMetal()* workload, for the bare-metal setup, and the *RiscvLinux()* workload, for regular FS simulation, from the command line terminal. We developed a configuration file that supports both bare-metal setup and FS mode with full OS support, based on gem5's *fs_linux.py* script. The details of our configuration file are described in later sections. We run Dhrystone both on bare-metal setup, to obtain the IPC, and regular FS mode, to obtain the Dhrystone score in DMIPS/MHz.

When running the FS configuration, we use a pre-built Linux kernel binary [43] (riscv-bootloader-vmlinux-5.10) alongside a simple RISC-V disk image based on *BusyBox* [44], obtained from *gem5-resources* repository [45].

4.4.1 System object and configuration script setup

First of all, file *ariane_fs.py* is created and it defines the system object used in the configuration. Class *ArianeSystem* is created and it inherits from gem5's *System* class. This class contains all of the necessary modules for the FS mode. To initialize the system, the class constructor needs to be defined. The constructor creates all the modules needed for the simulation and initializes the system. It takes four parameters:

- *kernel*: kernel binary path.
- *disk*: disk image path.
- *num_cpus*: Number of CPUs.
- *bare_metal*: Run simulation in bare-metal mode.

These can be passed and configured from the command line options. We provide both the flow and commands in Appendix A.

The constructor function defines the system's clock as well as the system's memory range, which is the size of our physical memory. The system bus is also created in the constructor method. For the system bus, gem5's *MemBus* module was used. This module uses the *SystemXBar* with a *Bad address responder* added. The *Bad address responder* is a fake device that returns a bad address error on any access. The system port is then set to this bus and the *Hifive()* platform is set as the system's platform.

Next, if the *bare_metal* option is true, the system's workload is set as *RiscvBareMetal()* and the *bootloader* parameter is set as the *kernel* option parameter. This way the simulation runs on a bare-metal setup, thus without OS support.

If the *bare_metal* option is false, the system's workload is set as *RiscvLinux* and the *obj_file* parameter is set as the *kernel* option parameter and the simulation proceeds to run a regular Full System simulation. In this mode, the required Device Tree (DTB) file is created and set using method *generateDtb()* from gem5's *fs_linux.py* and the kernel parameters are set in the *kernel_cmd* parameter.

The rest of the constructor function calls a number of helper functions that initialize the system's modules. Specifically:

- ***createCPU(num_cpus)***: This method, takes the number of CPUs from the command line options and sets the system's CPU parameter as module *Ariane()*, which is created in file *ariane.py*. The memory mode parameter is set in *'timing'* mode and gem5's method *createThreads()* is called to create hardware threads for the system.
- ***createCacheHierarchy()***: In this method, modules *Ariane_icache()* and *Ariane_dcache()* from file *ariane.py* are used to set the systems instruction and data cache respectively and connect them to the CPU and the memory bus. Furthermore, the CPU's MMU is set using gem5's *RiscvMMU()* module and the instruction and data TLB sizes are defined. The MMU's walker ports are set to the memory bus *cpu_side_ports*.
- ***createInterrupts()***: This method creates the systems interrupts by calling gem5's *createInterruptController()* method.
- ***createMemoryController()***: The memory controller is set in this method. gem5's *DDR3_1600_8x8* module with the memory range defined above is used in the *dram* parameter and the memory bus side ports in the *port* parameter.
- ***createDevices(disk)***: This method sets up all the peripheral devices used in the system. It defines the system's I/O bus using *IOXBar()* and real-time clock using *RiscvRTC()* and connects the I/O bus and the memory bus to the system's bridge. Moreover, a disk image object is created using the disk image given in the command line options parameter. Core-local (CLINT) and Platform-level (PLIC) interrupt controllers are set and on-chip and off-chip I/O are connected to the memory and

I/O buses. Finally, the MMU's physical memory address checker is set using gem5's *PMAChecker()* module.

Running the configuration file *ariane_fs.py* will create and simulate the system shown in Figure 4.8.

4.4.2 Creating a run script

To be able to run a Full System simulation for the above *System* object, we create the script *ariane_fs_run.py*. The system mentioned above is imported from *ariane_fs.py*.

The script reads the command line options, using the *argparse* library, and passes them as arguments to the simulated system. Using gem5's *Options* library the script supports all of gem5's common FS mode command line options.

After defining the root object the script runs the simulation using the *run()* method from gem5's *Simulation* library.

4.4.3 Running the simulation

To run the Full System simulation, we need to run *gem5.opt* binary and call the configuration file. To run file *ariane_fs_run.py*, the kernel, the disk image with the file system and the number of cores have to be defined in the command line options. To run on bare-metal mode, option *–bare-metal* has to be added and the benchmark binary has to be defined in the command line options instead of the kernel.

When running the simulation in FS mode, standard output is not automatically redirected to the console. In order to communicate with the system, gem5 offers a serial terminal named *m5term*. After building *m5term*, to connect to the system we have to run the *m5term* binary, in a separate command line window while the gem5 simulation is running, and connect to port *3456* as a *localhost*.

After connecting to the system we can see Linux booting in the *m5term*. Inserting username and password *root*, we enter as a user in the command prompt where we can run Linux commands and execute benchmarks mounted on the disk image.

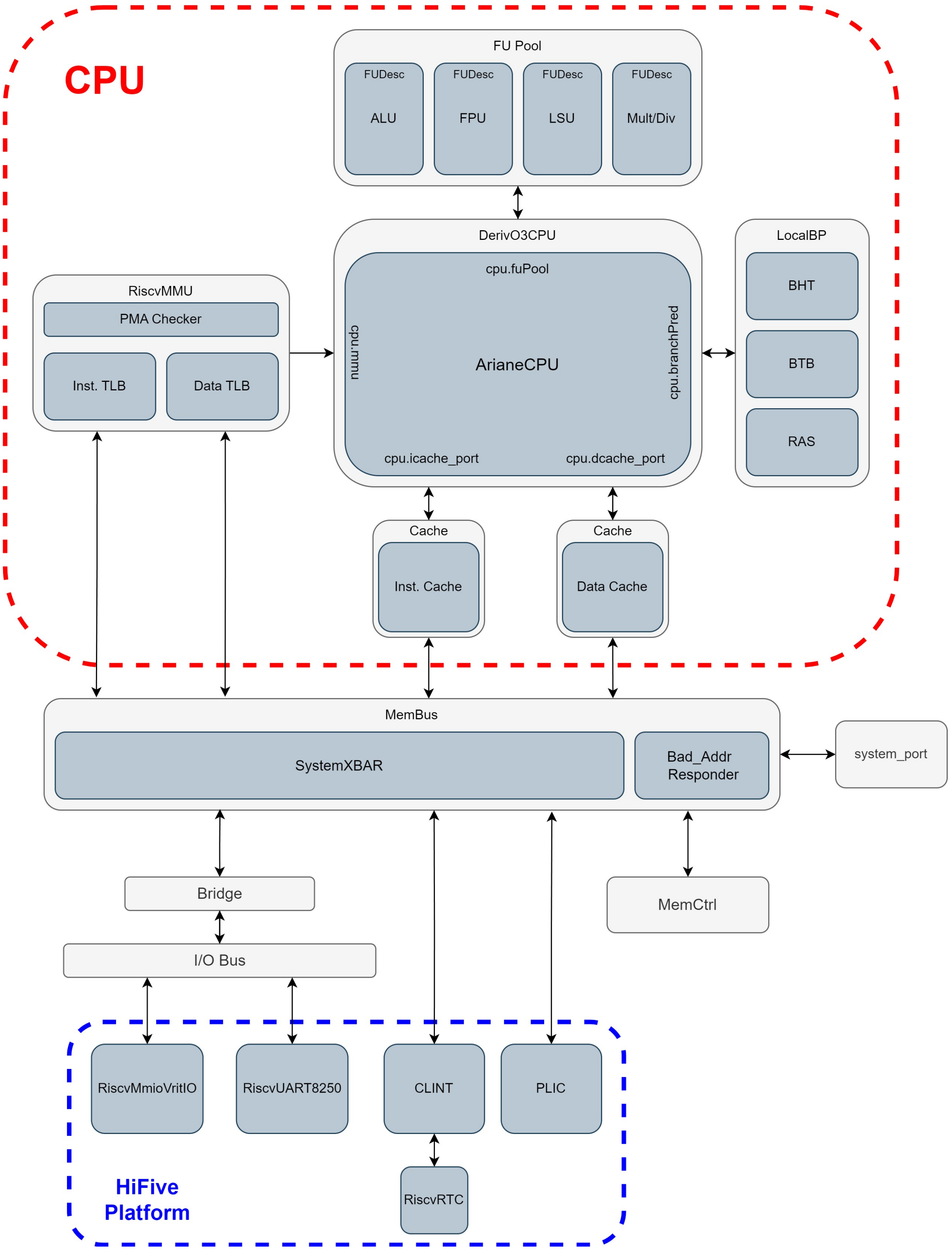


FIGURE 4.8: System configuration for Full System simulation in gem5.

In the case of bare-metal mode, the simulation keeps running after the benchmark has finished its execution. Since *m5term* is not supported in bare-metal mode, debug flag *Exec* is used, starting from the beginning of the simulation. This debug flag turns on instruction tracing in gem5 and it makes the simulation print a disassembled version of each instruction along with other useful information after the execution is finished. File *trace.out* is defined and used to store the information printed from the debug flag.

Using file *trace.out* we notice that after the benchmark finishes the execution the simulation jumps in an infinite loop to signal its termination. Tests from the *riscv-tests* are built for bare-metal systems and communicate with a host machine to inform the result. The test, after the execution, expects an address from the host machine. gem5 is unaware of such address, thus the simulation execution never stops.

A possible solution to this problem would be to use function *m5_exit()* from the *m5ops* after the execution of the test. When this operation is called, it triggers the simulation to stop in nanoseconds specified by the user. However, tests from *riscv-tests* suite run in machine mode where *m5ops* are not supported.

To fix this issue, gem5's max instruction count was used. Using the trace file *trace.out*, after running the tests once, we can find the instruction where the test finishes its essential execution and starts the endless loop. We can use this threshold as the max instruction count, after which the simulation will exit.

4.5 Setting up McPAT for power and energy modeling

To recreate power and energy performance metrics, a modified version of McPAT, called cMcPAT, was used. To run McPAT, we have to pass an input *xml* file that contains micro-architectural design parameters and event statistics. Example *xml* input files are provided with McPAT. cMcPAT comes with helper scripts and a template *xml* file in order to produce energy metrics and create an input file for McPAT using gem5's output files.

4.5.1 Creating the input file for McPAT

First of all, we have to enter all known micro-architectural design parameters to the template *xml* input file. Script *GEM5ToMcPAT.py* parses files *stats.txt* and *config.ini* from gem5's output and inserts all useful design parameters and event statistics needed for McPAT, to the template *xml*. After running the script and passing the files: *stats.txt*, *config.ini* and the template *xml*, a new *xml* input file is produced that we can then use to run McPAT.

4.5.2 Running McPAT

Using the input *xml* file mentioned above, we run McPAT's binary passing the input file in the command line options. To get the most detailed output from McPAT, variable *-print_level* has to be set at level 5, which is the maximum detail level, and defined in the command line options. McPAT's output is then redirected into an output text file, from where we can examine its outcomes.

Chapter 5

Results and Performance Analysis

In this chapter, we describe the experimental procedure of matching the performance and energy costs of the silicon implementation of CVA6 (Ariane), to the gem5 configurations described in the previous chapters. We present the results on performance, leakage and dynamic power and energy, and provide a preliminary error analysis of the gem5 simulator, by comparing our results to the ones reported in [2].

5.1 Performance Experiments and Results

We use the configuration described in the previous chapter as our base model and attempt to match the performance of the silicon implementation of Ariane by tweaking the unknown parameters of the base implementation and keeping all the known parameters unmodified.

We create four models of the base configuration, *config0-config3*, with *config0* being the base configuration described in Chapter 4 and the rest being versions of the base configuration with modified parameters. In each model, we modify the values of *fetchQueueSize*, the load and the store queue size, *LQEntries* and *SQEntries*, and instruction queue size, *numIQentries*. For each model, we provide a secondary configuration, *configNb, with N the number of the model*, with an alternative branch prediction unit setup, since the authors of [2] report an alternative configuration of a BHT of 128 entries and a BTB of 64 entries. We provide the parameters of each model in the following table. We kept the rest of the unknown parameters unmodified, with the values described in the previous chapter. Table 5.1 indicates the parameter values of each configuration.

TABLE 5.1: Parameter values of all configurations tested.

Parameters	**config. 0**		**config. 1**		**config. 2**		**config. 3**	
BP config.	a	b	a	b	a	b	a	b
fetchBufferSize(bytes)	4	4	4	4	4	4	4	4
fetchQueueSize(entries)	4	4	4	4	8	8	16	16
LQEntries	2	2	4	4	8	8	16	16
SQEntries	2	2	4	4	8	8	16	16
numIQEntries	8	8	8	8	8	8	16	16
numROBEntries	8	8	8	8	8	8	8	8
localPredictorSize	8	128	8	128	8	128	8	128
localPredictorBits	2	2	2	2	2	2	2	2
BTBEntries	8	64	8	64	8	64	8	64
RASSize	2	2	2	2	2	2	2	2

In [2], the authors report an IPC of 0.82 and 1.65 DMIPS/MHz produced by the *Dhrystone* benchmark, using the secondary branch prediction unit configuration of a 128-entry BHT and a 64-entry BTB, however, they do not provide any additional information on how the benchmark was executed or if the configuration that produced that number has further differences than the branch prediction unit. To obtain the IPC of our configurations, we decided to run the Dhrystone benchmark, for each configuration described above in bare-metal setup for 200k instructions, since we assume that bare-metal mode will produce a more representative IPC result. After a test run of Dhrystone, we noticed that the *exit* environment call is executed nearly at 200k instructions, therefore we chose an instruction limit of 200k instructions. Since gem5 does not support the serial terminal in bare-metal mode, we executed the Dhrystone benchmark in Full-System mode, to obtain the *Dhrystones per second* score in order to calculate the DMIPS/MHz score.

Since the division unit does not have a fixed latency in Ariane, we execute the benchmark for different division latency values in each configuration shown in Table 5.1, to see if and how it affects performance. We simulate each configuration four times, with 2 and 64-cycle latency, for best and worst cases, and for 16 and 32-cycle latency for middle cases.

5.1.1 IPC Results from the Dhrystone Benchmark

Running the Dhrystone benchmark through all configurations indicated in Table 5.1, and all the selected division latency values in bare-metal mode for 200k instructions, produces the IPC results shown in the following table. We obtain the IPC of each configuration the gem5's *stats.txt* output file. Instructions per cycle is indicated by the *system.cpu.totalIPC* statistic.

TABLE 5.2: IPC results from the Dhrystone benchmark in bare-metal simulation.

Div (cycles)	2	16	32	64
Configuration	**IPC**			
0a	0.439027	0.437082	0.429139	0.41473
0b	0.470416	0.463891	0.454975	0.439043
1a	0.576997	0.572789	0.560634	0.535748
1b	0.648076	0.63551	0.618887	0.590159
2a	0.616624	0.61187	0.596495	0.569718
2b	0.701728	0.686034	0.667607	0.633484
3a	0.62172	0.617351	0.602237	0.574094
3b	0.704613	0.691017	0.672168	0.63773

Running the Dhrystone benchmark on same configurations as before, for all chosen division latency values, but in Full-System mode this time produces the following results (Table 5.3). For the FS mode runs we chose to run Dhrystone for 10 million runs, since, through test runs, we discovered that it did not produce the *Dhrystones per second* score for less than that. The Dhrystone benchmark, at the end of its execution, produces a score named *Dhrystones per second*. To calculate DMIPS/MHz from the Dhrystone score we use the formulas written below. After running the benchmark, we notice that for each configuration, each run with different division latency does not affect Dhrystone's score, thus the result stays the same with each run. Therefore, in Table 5.3 the result of each configuration was produced on all individual runs with different division latency values.

$$DMIPS = \frac{Dhrystones\ per\ second}{1757},$$

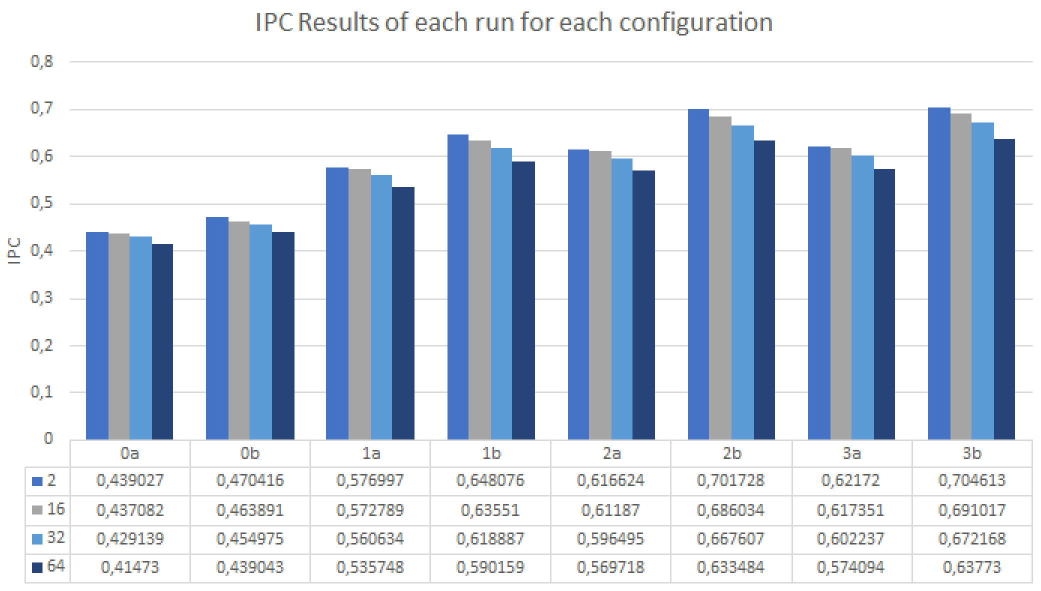
$$DMIPS/MHz = \frac{DMIPS}{clock\ speed(MHz)}$$

To calculate *DMIPS* we divide the *Dhrystones per second* by 1757. The number 1757 is the Dhrystone score produced by *VAX 11/780* which is nominally an *1 MIPS* machine. Dividing *DMIPS* with the clock speed in MHz, normalizes *DMIPS* into *DMIPS/MHz*, enabling us to compare processor performance at different clock speeds.

TABLE 5.3: Dhrystones per second and DMIPS/MHz score from the Dhrystone benchmark in Full-System simulation.

Configuration	**Dhrystones/second**	**DMIPS**	**DMIPS/MHz**
0a	3333333	1897.17	1.115
0b	3333333	1897.17	1.115
1a	5000000	2845.76	1.67
1b	5000000	2845.76	1.67
2a	5000000	2845.76	1.67
2b	5000000	2845.76	1.67
3a	5000000	2845.76	1.67
3b	5000000	2845.76	1.67

The IPC results are presented in the following figure (fig. 5.1). In Figure 5.1, each group in the horizontal axis corresponds to each configuration. The "bars" correspond to the IPC result for each run with a different division latency. IPC value is indicated by the vertical axis.



	0a	0b	1a	1b	2a	2b	3a	3b
■ 2	0,439027	0,470416	0,576997	0,648076	0,616624	0,701728	0,62172	0,704613
■ 16	0,437082	0,463891	0,572789	0,63551	0,61187	0,686034	0,617351	0,691017
■ 32	0,429139	0,454975	0,560634	0,618887	0,596495	0,667607	0,602237	0,672168
■ 64	0,41473	0,439043	0,535748	0,590159	0,569718	0,633484	0,574094	0,63773

FIGURE 5.1: IPC and DMIPS/MHz results graph for all the configurations.

With a first glance at the results, we notice that the IPC is increased between each configuration and comes closer to the published result, as we increase the sizes of the load and store queues, the fetch queue and the instruction queue. Furthermore, as we increase the latency of the divisions, we notice that the performance drops. We can see that the increase of division latency in cycles has a more significant effect on configurations 2 and 3 than on configurations 0 and 1. Considering the FS simulation, we notice that configuration 0 produces 1.116 DMIPS/MHz while the rest configurations produce almost the double performance with 1.67 DMIPS/MHz, which matches almost perfectly the reported 1.65 DMIPS/MHz.

5.1.2 Performance comparison between configurations

To illustrate the impact on the performance of each configuration we present the following figures (fig. 5.2 and 5.3). Both figures present the resulting IPC of each configuration for each run, Figure 5.2 presents the performance of configurations with the original BPU setup while Figure 5.3 with the alternative BPU setup. In both figures, each group in the horizontal axis presents the division latency value in cycles, each "bar" indicates the performance of the corresponding configuration and the vertical axis the IPC value.

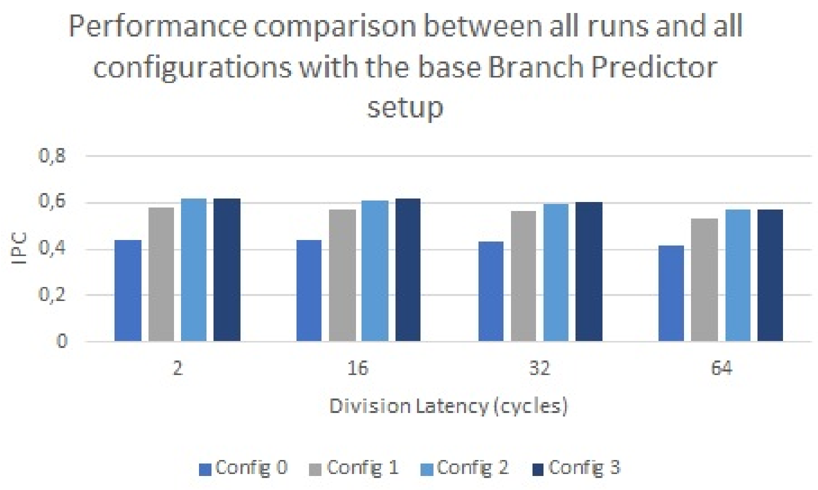


FIGURE 5.2: Performance comparison between configurations with the original Branch Prediction Unit Setup.

From both figures, we notice that configuration 1 has a more significant improvement compared to configuration 0, with the only difference between configurations being the load and store queue sizes. In addition, from configuration 2 we notice that increasing the load/store queue sizes along with the fetch queue size, further increases performance, however at a lower rate compared to configurations 0 and 1. Furthermore, we notice that despite configuration 3 having double the sizes of fetch, instruction and load/store queues, compared to configuration 2, it produces slightly increased but almost identical performance results.

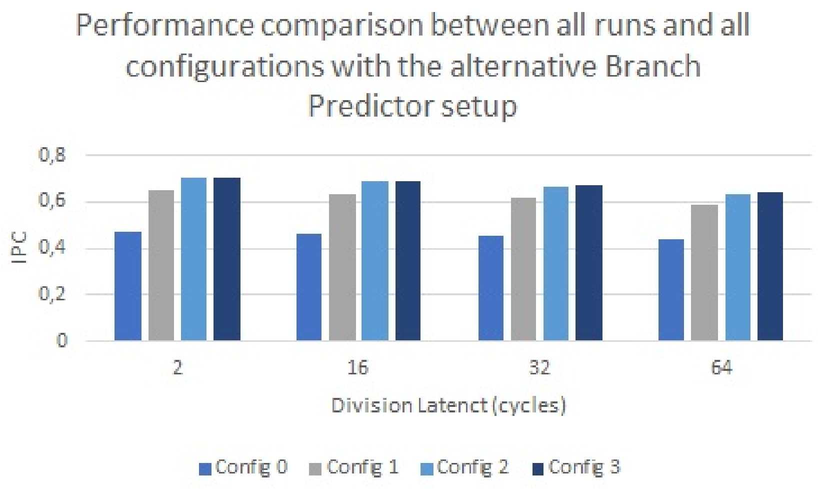


FIGURE 5.3: Performance comparison between configurations with the original Branch Prediction Unit Setup.

In the following figures (fig. 5.4 to 5.7), we present the results of each configuration respectively, in order to better illustrate the impact of the Branch Prediction setup, for each configuration. In each figure, the groups in the horizontal axis indicate the division latency value in cycles, the "bars" correspond to the BPU setup and indicate the performance in IPC with the corresponding value from the vertical axis.

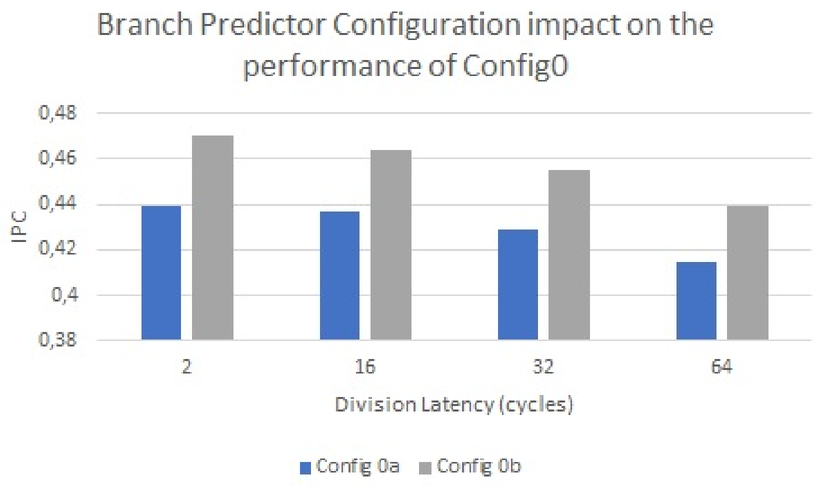


FIGURE 5.4: Performance results from all runs for Configuration 0, illustrating the impact of Branch Prediction Unit setup.

From figures 5.4-5.7, we notice that changing the BPU configuration singlehandedly offers a significant increase in the CPU's performance. For example, if we look at configuration 2 (fig. 5.6), *configuration 2a* produces a performance of 0.61 IPC when running with a division latency of 2 cycles.

Changing to the alternative BPU setup, in *configuration 2b*, the performance increases from 0.61 IPC to 0.70. We notice a similar increase in performance to the rest configurations respectively (fig. 5.4, 5.5 and 5.7)

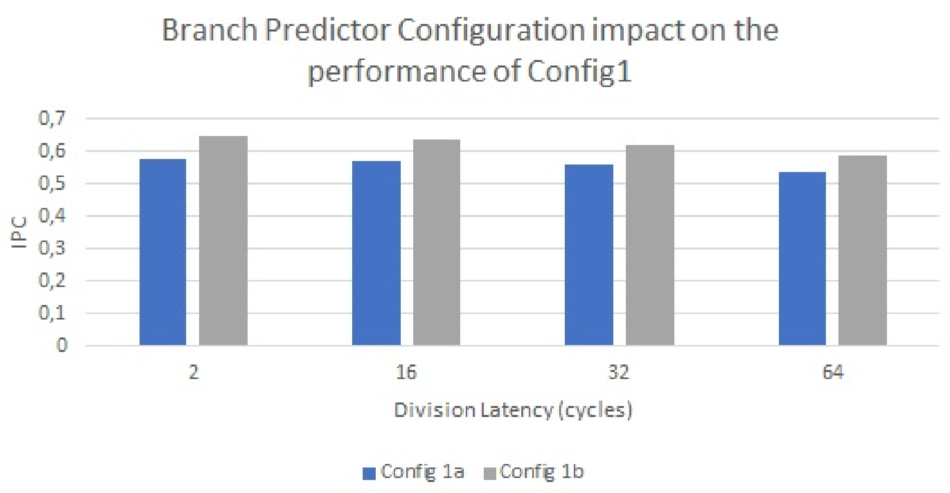


FIGURE 5.5: Performance results from all runs for Configuration 1, illustrating the impact of Branch Prediction Unit setup.

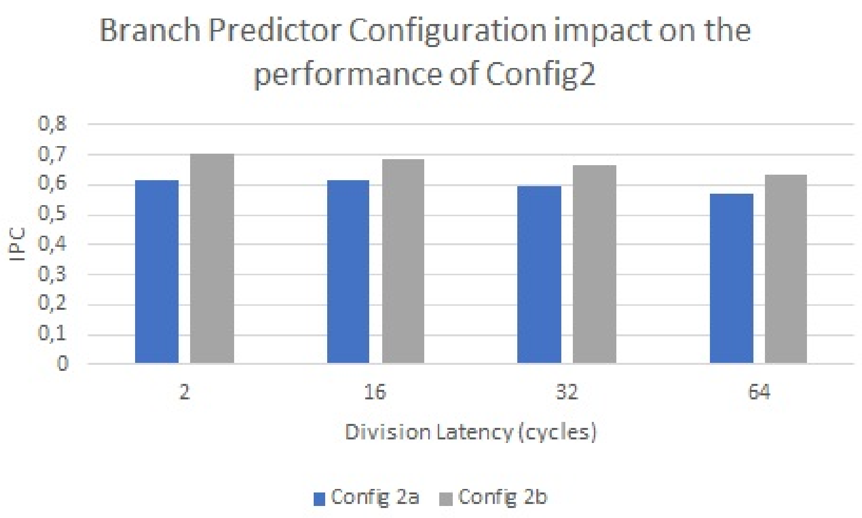


FIGURE 5.6: Performance results from all runs for Configuration 2, illustrating the impact of Branch Prediction Unit setup.

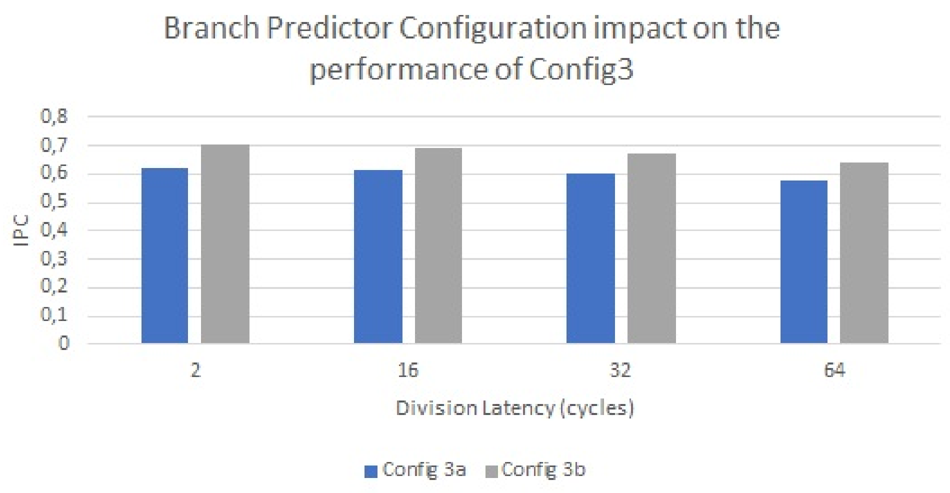


FIGURE 5.7: Performance results from all runs for Configuration 3, illustrating the impact of Branch Prediction Unit setup.

We also provide the following figures (Fig. 5.8 to 5.11), in order to make a clearer presentation of the impact of each configuration on performance. Each figure presents the performance of all configurations for a specific division latency value in cycles. The groups in the horizontal axis present each configuration and the "bars" indicate the change in branch prediction setup. The vertical axis indicates performance in IPC.

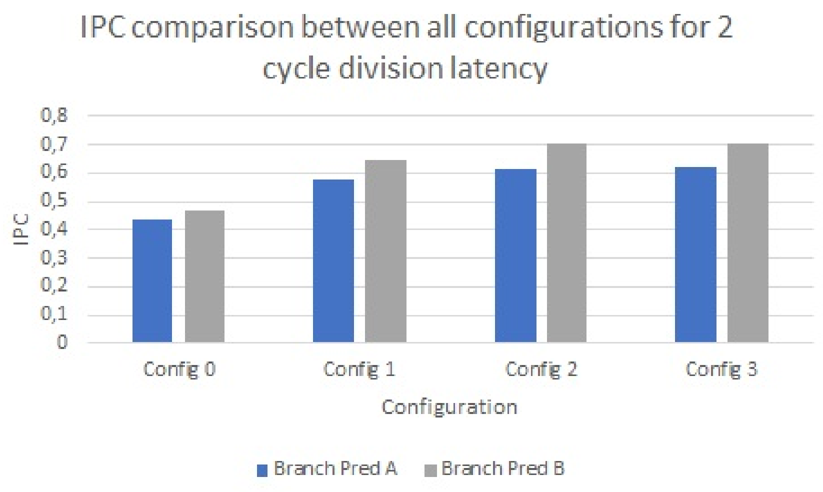


FIGURE 5.8: Performance comparison between all configurations for a division latency of 2 cycles.

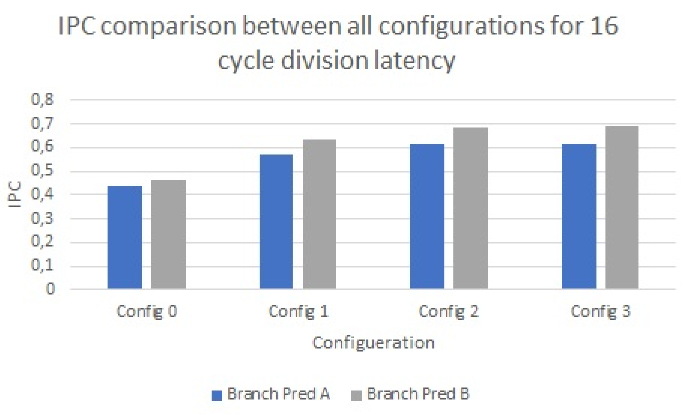


FIGURE 5.9: Performance comparison between all configurations for a division latency of 16 cycles.

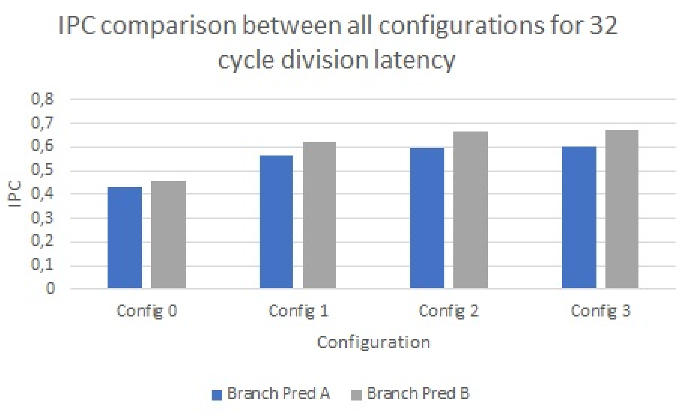


FIGURE 5.10: Performance comparison between all configurations for a division latency of 32 cycles.

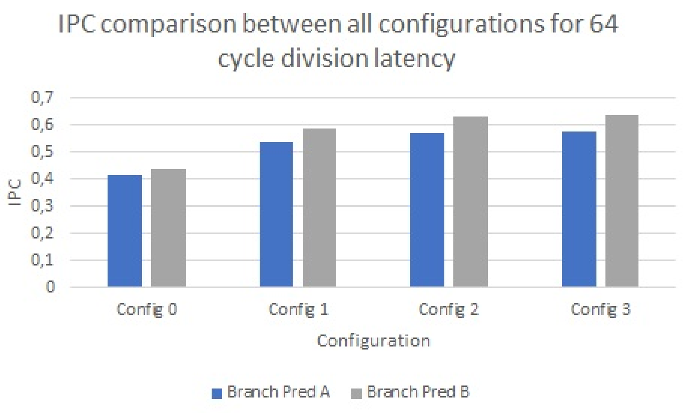


FIGURE 5.11: Performance comparison between all configurations for a division latency of 64 cycles.

Through all the figures above (fig. 5.8-5.11), we notice that each configuration has improved in performance from the previous one, however, configuration 3 produces almost identical performance to configuration 2, despite the former having double the size of fetch, instruction and load/store queues. Moreover, throughout all configurations, the alternative branch prediction configuration offers a substantial increase in performance.

5.1.3 Matching the reported performance

From the results reported above (Tab. 5.2), we notice that configuration 2 and configuration 3 produce almost identical performance, despite the latter having double the size of fetch, instruction and load/store queues. If we further increase the values of these design parameters (e.g. 32, 64-entry queues), the simulation will produce a similar performance to configurations 2 and 3. A possible cause for this could be that the system becomes bottlenecked from the lower values of the known design parameters. After fine-tuning the parameters of configuration 3b, the performance of which comes closer to the reported, we notice that increasing the size of the Reorder Buffer from 8 to 12 entries, produces a performance of 0.81 IPC, for a division latency of 2 cycles, matching almost perfectly the published 0.82 IPC of Ariane. Figure 5.12, presents the comparison of the new configuration, 3c, with configurations 2 and 3. Although this configuration matches the performance of Ariane, it is not justifiable, since it does not match the specifications of Ariane's circuit.

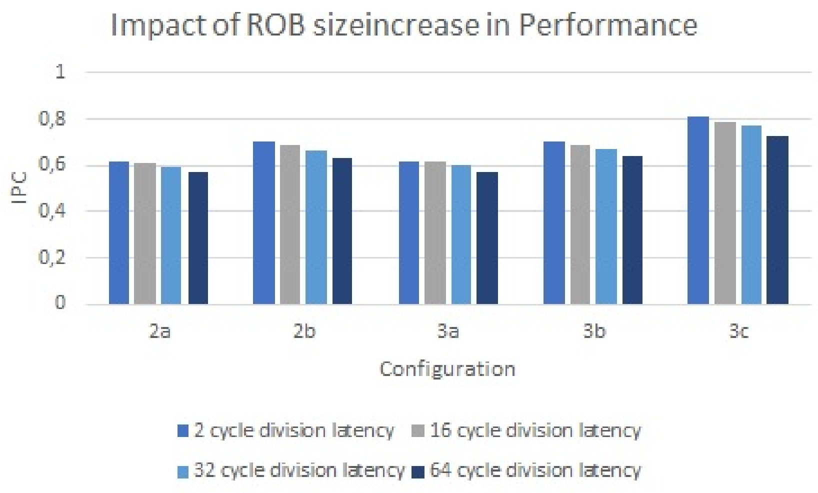


FIGURE 5.12: Performance comparison between configuration 3c and configurations 2 and 3.

5.1.4 Discussion

From the results presented above, first of all, we notice that the division latency shows a greater impact in configurations 2 and 3 than in 0 and 1. In configurations 2 and 3, the sizes of the CPU's subsystems are large enough not to affect performance, thus, performance is more dependent on the divider unit. Contrariwise, in configurations 0 and 1, the performance is limited by the small subsystems of the CPU making the divider unit affect the performance in a lesser scale.

Furthermore, in each configuration, we notice that the branch prediction unit configuration can greatly increase performance. In addition, we notice a great increase in performance from configuration 0 to configuration 1, where we increase the size of the load/store queue, i.e. when we modify components that affect the memory system. However, we cannot draw clear conclusions, since we do not have access to further information and base performance statistics regarding the memory system of Ariane. Moreover, a memory-intensive benchmark would be better suited to further examine the impact of gem5's memory system on performance, compared to the Dhrystone benchmark, since Dhrystone is a compute-intensive benchmark.

Unfortunately, we were not able to match the reported performance with the base configurations we presented at the beginning of this chapter. From configurations 2 and 3, we notice that while doubling the size of key modules of the CPU, the performance remained almost the same. Increasing further the values of the unknown parameters of our configurations, the performance would reach a ceiling. We can argue that in this case the system is bottlenecked by the small values of the known design parameters, and the performance could not be further increased only by modifying the unknown parameters. By slightly increasing the Reorder Buffer entries in *config3b*, the performance of which is the closest to the reported, we produce a performance of 0.81 IPC matching almost perfectly the reported performance of 0.82 IPC. However, this configuration is greatly differentiated from the silicon implementation of Ariane.

5.2 Power and Energy Experiments and Results

To obtain the power and energy costs of our simulated system, we used the *McPAT* power simulator and specifically a modified version of it called *cMcPAT* which improves upon accuracy issues of the standard version of McPAT reported in [24] and offers scripts that integrate gem5 with McPAT. To match the workload used to produce the energy costs and leakage power in [2], we selected the following tests and benchmarks from the *riscv-tests* suite:

- **ALU instructions:** *rv64ui-p-add*
- **Multiplications:** *rv64um-p-mul*
- **Divisions:** *rv64um-p-div*
- **Load and Stores without virtual memory enabled:** *rv64ui-p-sw*
- **Load and Stores with virtual memory enabled:** *rv64ui-v-sw*
- **Mixed workload:** Matrix multiplication benchmark, *mm.riscv*

The assembly, *ISA*, tests from the suite, are not designed to measure performance and power since their code is too short. Therefore, we modified their source code to loop the main testing code for 200 iterations. We will simulate the selected benchmarks, on the configurations described above, on gem5's bare-metal mode to obtain performance statistics. Then, we will use McPAT to obtain the leakage and dynamic power of each run. Finally, using leakage and dynamic power from McPAT and total runtime from gem5 we will calculate the energy cost of each configuration using the formula written below.

$$Energy = (Total\ Leakage + TotalDynamic) * runtime$$

From the selected benchmarks, division latency only affects *rv64um-p-div* and *mm.riscv*. The rest of the benchmarks are not affected since no *div* instruction is executed. Therefore, we will execute *rv64um-p-div* and *mm.riscv* for multiple runs, on each configuration, changing the division latency value in cycles, in each run. For each configuration, we will simulate these benchmarks for a division latency of 2, 16, 32 and 64 cycles.

5.2.1 Leakage Power Results

After executing all the selected benchmarks on both gem5 and McPAT, we obtain the *Subthreshold Leakge* and *Gate Leakage* power of each configuration. *Subthreshold Leakage Power* is the leakage power produced when the transistor is in the subthreshold region, meaning when the transistor is "off". Leakage power is affected by the transistor technology, temperature and circuit area. Table 5.4 presents the subthreshold, gate and total leakage power of each configuration obtained by McPAT, with total leakage being the sum of subthreshold and gate leakage.

TABLE 5.4: Leakage Power of each configuration on every benchmark in W.

Configuration	Sub. Leakage (W)	Gate Leakage (W)	Tot. Leakage (W)
0a	0,000145167	0,000226412	0,000371579
0b	0,000145241	0,000226505	0,000371746
1a	0,000145167	0,000226412	0,000371579
1b	0,000145241	0,000226505	0,000371746
2a	0,000145167	0,000226412	0,000371579
2b	0,000145241	0,000226505	0,000371746
3a	0,000145301	0,000226591	0,000371892
3b	0,000145375	0,000226684	0,000372059

Each configuration produces the same leakage power on every benchmark simulated since the transistor technology remains the same and the system is simulated for a constant temperature. From Table 5.4, we notice that configurations 0a to 2a and 0b to 2b all produce the same leakage power respectively, while configurations 3a and 3b produce a similar, though slightly increased leakage power. We assume that the changes in the circuit area of configurations 0 through 2 are not significant enough for McPAT, to affect leakage power, thus producing the same value. Moreover, increasing the size of the Branch Prediction Unit slightly increases total leakage power consumption. Furthermore, we notice that the total leakage power produced is significantly lower, less than half, than the reported leakage power of 1.08 mW.

5.2.2 Dynamic Power and Energy Results

The *Dynamic Power* of each configuration for every benchmark run is presented below in Table 5.5. *Dynamic Power* is produced by capacitive power when charging or discharging transistors, at transitions from 0 to 1 and from 1 to 0, and by short-circuit power produced due to brief short-circuit current during transitions.

TABLE 5.5: Dynamic Power of each configuration on every benchmark in W.

Configuration	rv64iu-p-add	rv64um-p-mul	rv64um-p-div	rv64ui-p-sw	rv64ui-v-sw	mm.riscv
0a	0,03827	0,038175	0,043177	0,055576	0,0508388	0,06434
0b	0,037965	0,037879	0,043207	0,055092	0,0512379	0,064576
1a	0,041649	0,041793	0,048124	0,059014	0,053945	0,093622
1b	0,04128	0,041431	0,048158	0,058463	0,0543945	0,094095
2a	0,041658	0,041802	0,048072	0,059024	0,0543054	0,11608
2b	0,041289	0,041441	0,048106	0,058473	0,0547604	0,117058
3a	0,041744	0,041893	0,048336	0,059295	0,0547688	0,118281
3b	0,041373	0,041531	0,04837	0,058742	0,0552274	0,119184

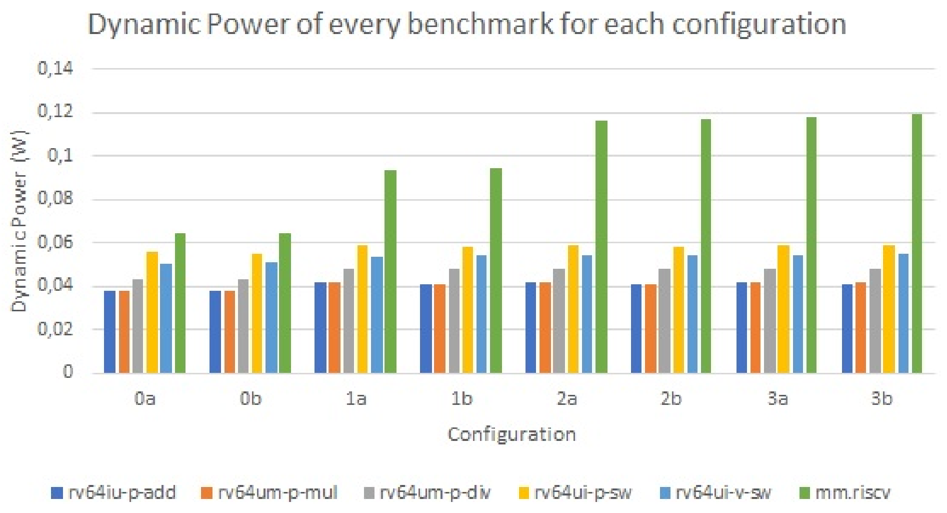


FIGURE 5.13: Dynamic Power results of each configuration on every benchmark.

Figure 5.13 illustrates the dynamic power consumption of each workload in each configuration. The groups in the horizontal axis correspond to each configuration while the bars in each group correspond to each benchmark. The vertical axis indicates dynamic power in W. We notice that each configuration consumes slightly more power than the previous one, however, the differences in power consumption between configurations remain minimal, with the exception of the *mm.riscv* benchmark, where the power consumption significantly increased when increasing the fetch, load/store and instruction queues. Furthermore, comparing configurations with the original BPU setup and the alternative BPU setup, we notice that increasing the size of the branch prediction unit slightly decreases dynamic power consumption, however it raises leakage.

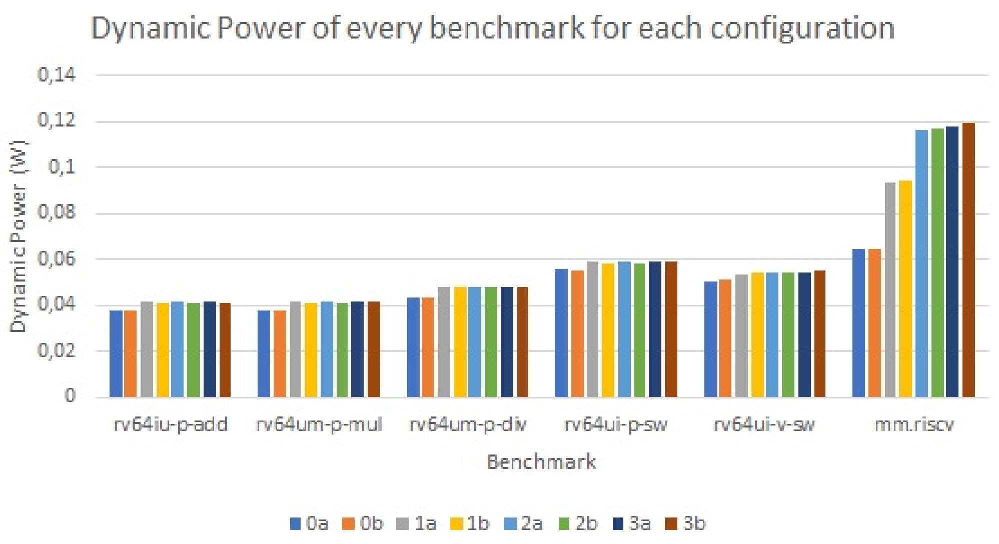


FIGURE 5.14: Dynamic Power results of each configuration on every benchmark, grouping the benchmarks.

Figure 5.14, presents the dynamic power consumption of each workload in each configuration, comparing the different workloads. The groups in the horizontal axis correspond to each workload while the bars in each group correspond to each configuration. The vertical axis indicates dynamic power in W. We notice that *mm.riscv* is the most demanding in dynamic power, where the impact of each configuration difference is noticeable, due to being the most complex, utilizing multiple resources of the CPU.

Multiplying the total power, obtained by adding the total leakage and dynamic power produced by McPAT, with the total runtime for each benchmark from gem5, we obtain the energy of each configuration for each benchmark as shown bellow, in Table 5.6.

TABLE 5.6: Energy of each configuration on every benchmark in J.

Configuration	rv64iu-p-add	rv64um-p-mul	rv64um-p-div	rv64ui-p-sw	rv64ui-v-sw	mm.riscv
0a	3,13E-06	3,35E-06	5,23E-07	4,48E-06	5,06E-06	2,12E-05
0b	3,14E-06	3,37E-06	5,23E-07	4,49E-06	5,05E-06	2,12E-05
1a	3,11E-06	3,33E-06	5,33E-07	4,51E-06	5,10E-06	2,11E-05
1b	3,12E-06	3,34E-06	5,34E-07	4,53E-06	5,09E-06	2,12E-05
2a	3,11E-06	3,33E-06	5,33E-07	4,51E-06	5,08E-06	2,11E-05
2b	3,12E-06	3,35E-06	5,33E-07	4,53E-06	5,07E-06	2,11E-05
3a	3,12E-06	3,34E-06	5,36E-07	4,53E-06	5,12E-06	2,11E-05
3b	3,13E-06	3,35E-06	5,36E-07	4,49E-06	5,11E-06	2,12E-05

Figure 5.15, presents the energy costs of each workload in each configuration. The groups in the horizontal axis correspond to each configuration while the bars in each group correspond to each benchmark. The vertical axis indicates energy in J. Throughout each configuration, we notice that the energy cost for each benchmark is similar.

We notice that the energy cost of *mm.riscv* is the highest since it was the most power demanding and its runtime was the longest among the rest benchmarks. Furthermore, the *rv64um-p-div* benchmark, although demanding in power, compared to the rest of assembly tests, costs the least in energy, since its runtime was the smallest.

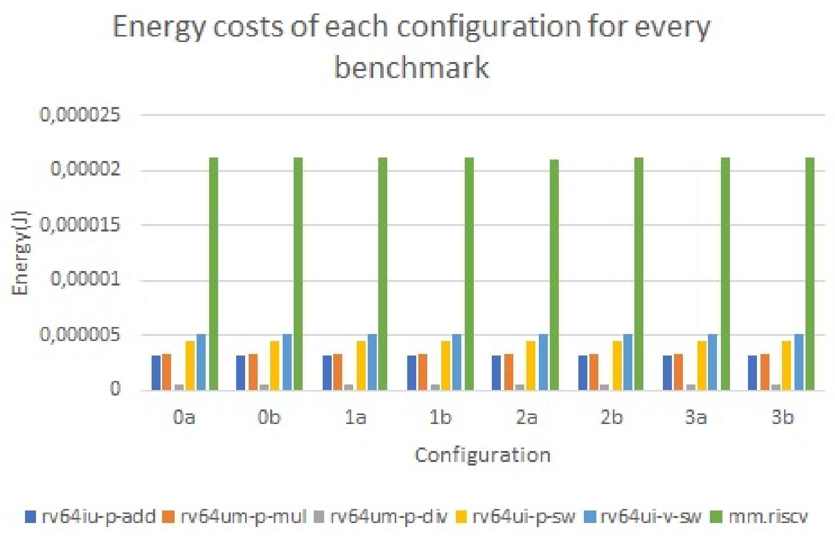


FIGURE 5.15: Energy costs of each configuration on every benchmark.

The energy costs between each workload are presented in Figure 5.16. The groups in the horizontal axis correspond to each workload while the bars in each group correspond to each configuration. The vertical axis indicates energy in J.

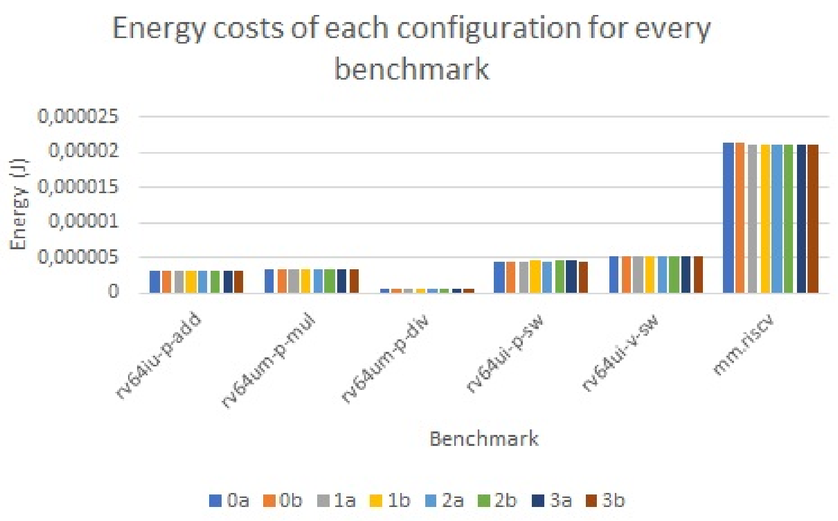


FIGURE 5.16: Dynamic Power results of each configuration on every benchmark, comparing the benchmarks.

Comparing our energy results to Ariane's published energy costs, we notice that the mismatch is unjustifiably immense. The energy costs produced by our simulated system are not justified, especially for a small CPU, such as Ariane's.

TABLE 5.7: Dynamic Power results of the rv64um-p-div test for different division latency values.

	Division latency in Cycles			
Configuration	2	16	32	64
0a	0,043177	0,022603	0,01332	0,007537
0b	0,043207	0,022618	0,013329	0,007542
1a	0,048124	0,023843	0,013719	0,00765
1b	0,048158	0,02386	0,013728	0,007655
2a	0,048072	0,023837	0,01372	0,00765
2b	0,048106	0,023853	0,013729	0,007658
3a	0,048336	0,023925	0,01376	0,007671
3b	0,04837	0,023942	0,01377	0,007676

Since the latency of Ariane's Divider unit is not fixed, we simulated the *rv64um-p-div* and *mm.riscv* benchmarks for different division latency values. Table 5.7 presents the dynamic power results of each run, with different division latency values, of each configuration, produced by the *rv64um-p-div* test. The dynamic power results of the *rv64um-p-div* test, are better illustrated in Figure 5.17. The groups in the horizontal axis correspond to each configuration while the bars in each group correspond to each run with different division latency. The vertical axis indicates the dynamic power result of each run in W.

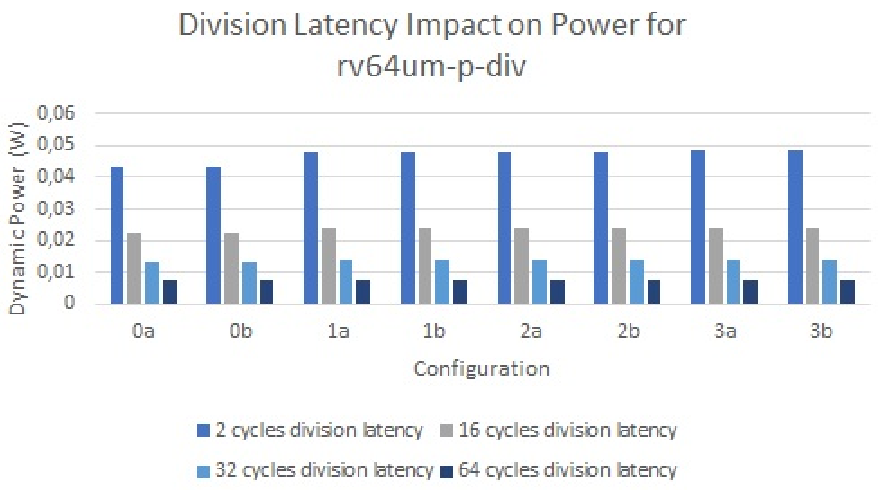


FIGURE 5.17: Dynamic Power results chart of the rv64um-p-div test for different division latency values.

Looking at Figure 5.17, first of all, we notice that the total dynamic power is increased in each subsequent configuration, i.e. as we increase the size of the fetch, load/store and instruction queue. Furthermore, the dynamic power of each configuration is almost halved as the division latency is increased.

TABLE 5.8: Energy results of the rv64um-p-div test for different division latency values.

	Division latency in Cycles			
Configuration	2	16	32	64
0a	5,23E-07	5,28E-07	5,48E-07	5,77E-07
0b	5,23E-07	5,29E-07	5,48E-07	5,78E-07
1a	5,33E-07	5,33E-07	5,35E-07	5,78E-07
1b	5,34E-07	5,33E-07	5,36E-07	5,78E-07
2a	5,33E-07	5,33E-07	5,35E-07	5,78E-07
2b	5,33E-07	5,33E-07	5,36E-07	5,78E-07
3a	5,36E-07	5,35E-07	5,37E-07	5,79E-07
3b	5,36E-07	5,35E-07	5,37E-07	5,79E-07

Table 5.8 presents the energy results of each individual run, with different division latency values, of each configuration, produced by the *rv64um-p-div* test. Respectively, the energy results are displayed in Figure 5.18. The groups in the horizontal axis correspond to each configuration while the bars in each

group correspond to each run with different division latency. The vertical axis indicates the energy result of each run in J.

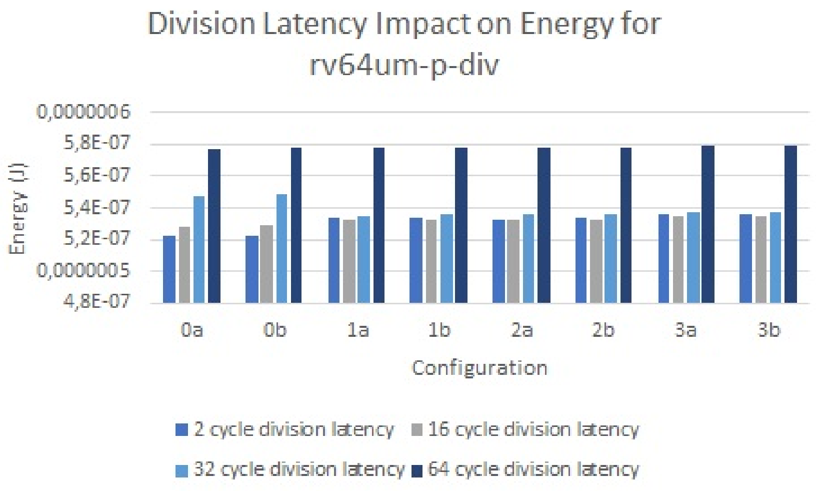


FIGURE 5.18: Energy results chart of the rv64um-p-div test for different division latency values.

In Figure 5.18, we notice that in each consecutive configuration, the energy is slightly increased. However, in contrast to the dynamic power, increasing the division latency, increases the energy consumption, since the total runtime is increased.

TABLE 5.9: Dynamic Power results of the mm.riscv benchmark for different division latency values.

	Division latency in Cycles			
Configuration	2	16	32	64
0a	0,06434	0,064294	0,064237	0,064149
0b	0,064576	0,064571	0,064511	0,06439
1a	0,093622	0,093533	0,093401	0,093139
1b	0,094095	0,094017	0,093873	0,093672
2a	0,11608	0,115938	0,11573	0,115315
2b	0,117058	0,116912	0,1167	0,116278
3a	0,118281	0,118139	0,117936	0,117512
3b	0,119184	0,119008	0,118819	0,118562

Respectively, the dynamic power of each configuration produced by the *mm.riscv* benchmark is presented in Table 5.9 and Figure 5.19. In Figure 5.19, the groups in the horizontal axis correspond to each configuration while the bars in each group correspond to each run with different division latency. The vertical axis indicates the dynamic power result of each run in W.

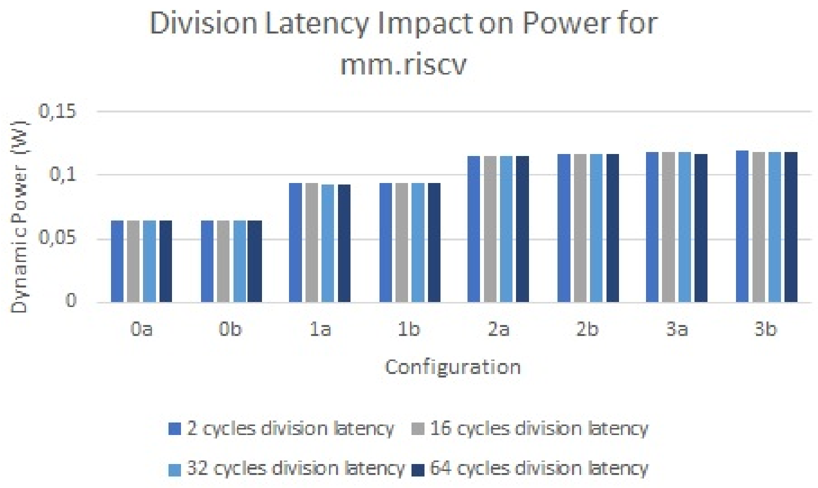


FIGURE 5.19: Dynamic Power results chart of the mm.riscv benchmark for different division latency values.

In each subsequent configuration, the dynamic power is increased, however, increasing the division latency, decreases the total power only by a little, since the *mm.riscv* benchmark uses multiple subsystems of the CPU and it is independent of the divider unit.

TABLE 5.10: Energy results of the mm.riscv benchmark for different division latency values.

	Division latency in Cycles			
Configuration	2	16	32	64
0a	2,12E-05	2,13E-05	2,13E-05	2,12E-05
0b	2,12E-05	2,12E-05	2,13E-05	2,12E-05
1a	2,11E-05	2,11E-05	2,11E-05	2,11E-05
1b	2,12E-05	2,11E-05	2,11E-05	2,12E-05
2a	2,11E-05	2,11E-05	2,11E-05	2,11E-05
2b	2,11E-05	2,11E-05	2,11E-05	2,11E-05
3a	2,11E-05	2,11E-05	2,12E-05	2,12E-05
3b	2,12E-05	2,11E-05	2,12E-05	2,12E-05

In addition, Table 5.10 and Figure 5.20, present the energy of each configuration produced by the *mm.riscv* benchmark. The groups in the horizontal axis, of Figure 5.20, correspond to each configuration while the bars in each group correspond to each run with different division latency. The vertical axis indicates the energy of each run in J.

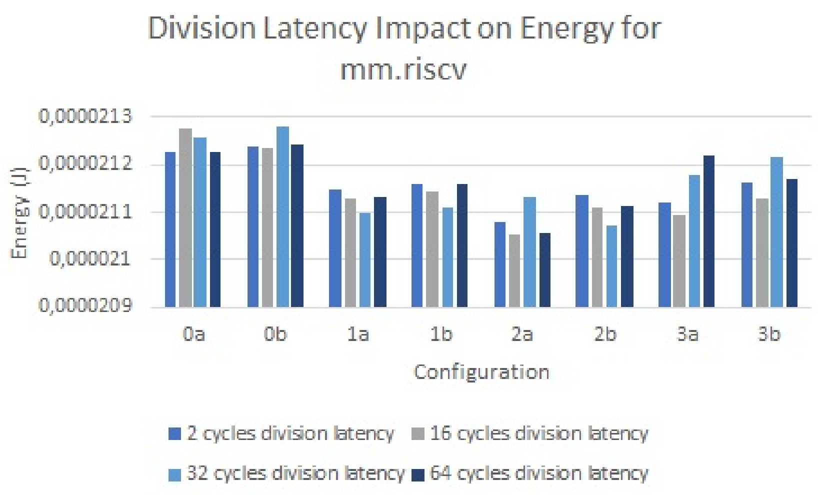


FIGURE 5.20: Energy results chart of the mm.riscv benchmark for different division latency values.

5.2.3 Discussion

First of all, we notice that the total leakage power produced, ≈ 0.37 mW, is significantly lower, less than half, than the reported leakage power of 1.08 mW. A possible cause for this mismatch could be that McPAT models and simulates CMOS technologies, while Ariane is implemented in 22nm FD-SOI transistor technology. In addition, McPAT has limited support for smaller CPUs, which causes the optimizer to crash or produce errors. Furthermore, since we do not have available information on Ariane's thermals, we run McPAT using a constant temperature of 340 K. Since leakage power is affected by the CPU's temperature, the absence of a more detailed approach regarding the thermals of our simulated system may lead to a higher mismatch in the produced total leakage power.

Furthermore, from the dynamic power results, we notice that in each subsequent configuration, the dynamic power is slightly increased in all benchmarks, with the exception of *mm.riscv*. In this case, the differences between configurations are clear, since the *mm.riscv* has a more complex workload, compared to the rest of the selected benchmarks, that utilizes multiple components of the CPU. In addition, comparing *rv64ui-p-sw* and *rv64ui-v-sw*, we notice that the former, i.e. the test that does not use virtual memory, consumes more power than the latter, which uses virtual memory. In this case, we would expect the test that uses virtual memory, to consume more power since it utilizes more components of the MMU and virtual address translation. Moreover, in *rv64um-p-div*, we notice that increasing the latency of the divider unit decreases the power consumption. Having a faster divider unit adds complexity to the circuit which increases power consumption. In general, we notice that the power consumption of our simulated system is much higher than expected. As a result, the total energy costs are also higher than expected, orders of magnitude higher compared to the reported, which is unacceptable for the scale and the simplicity of Ariane's circuit.

McPAT, has a number of well-known issues that affect the accuracy of its simulation, as reported in [24]. To decrease the mismatch of our results, we attempted to modify McPAT's models, unfortunately without any success since we did not fully discover the source of the error. Furthermore, there are available frameworks that calibrate McPAT's output to reduce accuracy errors, such as *McPAT-Calib* [28], [26], regarding the RISC-V architecture. Unfortunately, we were not able to use said frameworks as we lacked the data sets needed to train the framework's models.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In conclusion, in this thesis, we attempted to match the performance and energy costs of a silicon implementation of a RISC-V core, CVA6, to a simulated RISC-V system. We used the relatively new RISC-V framework of the gem5 simulator to implement our simulated system and produce performance statistics. We then integrate gem5 with a power simulator, McPAT, to obtain power and energy costs.

By fine-tuning the micro-architectural design parameter values of different configurations of our simulated system, we notice that the load/store queue and Re-order Buffer can have a significant impact on the system's total performance. We noticed that without modifying the known parameters the performance would reach a saturation point and further increase in the values of the unknown parameters would not produce the wanted performance. The configuration that matches the performance of Ariane has a much bigger load/store queue and Re-order Buffer. Therefore, we can conclude that the load/store queue and ROB can greatly affect the system's performance leading to accuracy loss.

Unfortunately, we did not manage to obtain meaningful power and energy results. McPAT's limited support on the RISC-V ISA and smaller cores causes a notable loss of accuracy. In addition, due to limited published design parameters and performance statistics, we were unable to use frameworks that calibrate McPAT, in order to decrease accuracy loss, such as McPAT-Calib [26]. We can conclude that McPAT was not capable of producing meaningful results without manually modifying its models.

6.2 Future Work

There is still a lot of work that can be done regarding the accuracy of the gem5 simulator in the context of RISC-V, especially when considering our conclusions. Hence, we provide the following suggestions for further research.

First of all, we encountered challenges in the modeling and evaluation stages of our system. Specifically, we could not obtain all the micro-architectural design parameters of Ariane and more importantly, we did not have any information on how the published performance and energy statistics were produced. This can lead to specification errors in the accuracy of our system since omitting important design parameters can affect performance and running the benchmarks under different circumstances (e.g different runtime, different compilation flags, bare-metal or on full OS) may lead to meaningless comparisons between our results and the published statistics. For future research, we suggest that Ariane should be implemented in an FPGA. This will allow us to manually execute benchmarks both on Ariane and the simulated system. This way, we can obtain more fine-grained performance statistics and we can isolate components of the CPU and identify causes of accuracy loss. Moreover, we suggest that we further investigate the Load/Store queue and Re-order Buffer to verify their impact on simulation accuracy.

Since we could not obtain meaningful power and energy results, we suggest further research on McPAT's support for RISC-V cores. In particular, we suggest further research on McPAT's back-end, to fully identify modeling errors from McPAT's models and adapt them to the target system. Moreover, using a CPU with more available performance statistics can enable the utilization of frameworks that calibrate McPAT, such as *McPAT-Calib* or *Powertrain*, which unfortunately we could not use in this study. Last, gem5 offers a premature power model, currently supported only for the ARM instruction set. Porting gem5's power model from ARM to RISC-V could be a great addition to gem5's RISC-V support and could simplify the process of power modeling using the gem5 simulator.

Appendix A

Environment Setup and Running the Scripts

In this section, we provide a brief guide on how to set up the environment, run the scripts and reproduce the results. We assume that *gem5* and *git* are already built, if not, the following link in [46] provides a guide on building gem5 and gem5's prerequisites including git. In this project, we use gem5 version 21.2.1.

A.1 Building the RISC-V GNU compiler toolchain

The RISC-V GNU compiler toolchain is provided in the following GitHub repository [47]. It includes the RISC-V C and C++ cross-compiler and supports a generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain.

First of all, make sure to install all the prerequisites listed in the repository's *README* section, depending on your operating system.

To obtain the source code clone the repository from the provided link [47]. Next, using a terminal, move to the toolchain's directory. To build type:

```
$ ./configure --prefix=/opt/riscv --enable-multilib
$ make Linux
```

Make sure that you include the `--enable-multilib` flag. If you want to build the toolchain for a different environment or a more custom build, additional information is provided repository's *README* section.

A.2 Building the RISC-V tests

The RISC-V tests suite can be obtained from the following GitHub repository [48]. Before building the tests, make sure that the RISCV environment variable is set the RISC-V tools install path. To obtain and build the tests, type in any working directory:

```
$ git clone https://github.com/riscv/riscv-tests
$ cd riscv-tests
$ git submodule update --init --recursive
$ autoconf
$ ./configure --prefix=\$RISCV/target
$ make
$ make install
```

After building, benchmark and test binaries will be generated in the *benchmarks* and *isa* directories respectively.

A.3 Obtaining the kernel binary and disk image from gem5-resources

First of all, gem5-resources can be obtained from the following GitHub repository [49]. To obtain the resources, type in any working repository:

```
$ git clone https://github.com/gem5/gem5-resources
```

Further information about the gem5-resources is provided in [45] and [49].

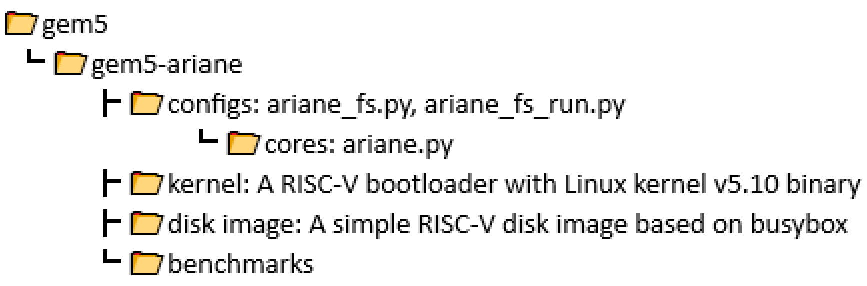
In this project, resources *riscv-disk-img*, which contains a simple pre-built RISC-V disk image based on busybox, and *riscv-bootloader-vmlinux-5.10*, that contains a pre-built RISC-V bootloader and Linux kernel v. 5.10, were used.

The resources can either be obtained from the links provided in the *resources.json* file contained in the *gem5-resources* directory, specifically [43] for the kernel and [44] for the disk image, or from the gem5 Resources website [50].

If you want to manually build the kernel and disk image, or use a different one, further information is provided in the following link [51]. A tutorial on creating disk images and mounting binaries in the disk image is provided in [52].

A.4 Running the Scripts

First of all, move the provided folder inside gem5's directory. The file structure should look like the one shown in Figure A.1.



```
gem5
└ gem5-ariane
    ├ configs: ariane_fs.py, ariane_fs_run.py
    │   └ cores: ariane.py
    ├ kernel: A RISC-V bootloader with Linux kernel v5.10 binary
    ├ disk image: A simple RISC-V disk image based on busybox
    └ benchmarks
```

FIGURE A.1: File structure of the experimental environment.

Copy and move file *ariane.py* from the directory `gem5/gem5-ariane/configs/cores` to the directory `gem5/configs/common/cores/riscv`. Next, copy and move files *ariane_fs.py* and *ariane_fs_run.py* from directory `gem5/gem5-ariane/configs` to directory `gem5/configs/example/riscv`.

To run the scripts, use the following command format, in the command line terminal. Make sure that you are inside gem5's directory.

```
/gem5$ ./build/RISCV/gem5.opt -d [gem5 output file path] [debug flags]
[config script path] [kernel binary path] [disk image path] [number of cores]
[optional bare metal flag] -I [max instruction number]
```

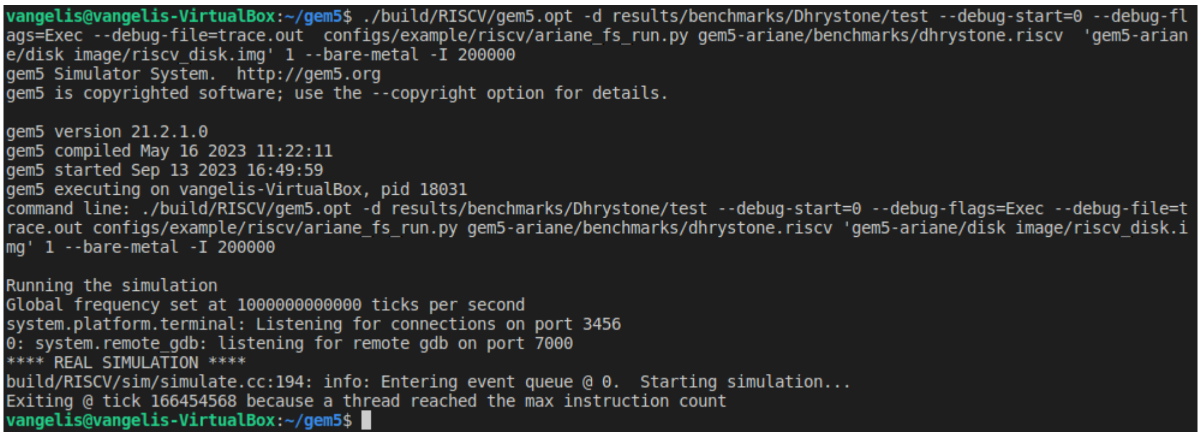
Running in Bare-Metal mode

To run in bare metal mode, include the `--bare-metal` flag and specify the benchmark binary in the "kernel binary" field. In bare metal mode, it is helpful to enable the *Exec* debug flag that enables instruction tracing and saves all executed instructions in a trace file. To enable the *Exec* debug flag, add `--debug-start=0 --debug-flag=Exec --debug-file=trace.out` in the "debug flags" field.

For example, to run the *Dhrystone* benchmark in bare metal mode, with one core, for 200000 instructions, run the command:

```
/gem5$ ./build/RISCV/gem5.opt -d results/benchmarks/Dhrystone/test
--debug-start=0 --debug-flags=Exec --debug-file=trace.out
configs/example/riscv/ariane_fs_run.py gem5-ariane/benchmarks/dhrystone.riscv
'gem5-ariane/disk image/riscv_disk.img'
1 --bare-metal -I 200000
```

After running the command, gem5 produces the following output in the command line terminal.

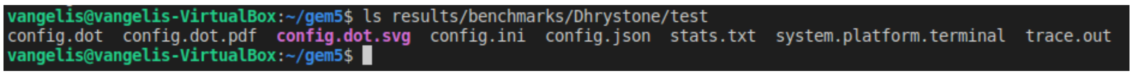


```
vangelis@vangelis-VirtualBox:~/gem5$ ./build/RISCV/gem5.opt -d results/benchmarks/Dhrystone/test --debug-start=0 --debug-fl
ags=Exec --debug-file=trace.out  configs/example/riscv/ariane_fs_run.py gem5-ariane/benchmarks/dhrystone.riscv  'gem5-arian
e/disk image/riscv_disk.img' 1 --bare-metal -I 200000
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.2.1.0
gem5 compiled May 16 2023 11:22:11
gem5 started Sep 13 2023 16:49:59
gem5 executing on vangelis-VirtualBox, pid 18031
command line: ./build/RISCV/gem5.opt -d results/benchmarks/Dhrystone/test --debug-start=0 --debug-flags=Exec --debug-file=t
race.out configs/example/riscv/ariane_fs_run.py gem5-ariane/benchmarks/dhrystone.riscv 'gem5-ariane/disk image/riscv_disk.i
mg' 1 --bare-metal -I 200000

Running the simulation
Global frequency set at 1000000000000 ticks per second
system.platform.terminal: Listening for connections on port 3456
0: system.remote_gdb: listening for remote gdb on port 7000
**** REAL SIMULATION ****
build/RISCV/sim/simulate.cc:194: info: Entering event queue @ 0.  Starting simulation...
Exiting @ tick 166454568 because a thread reached the max instruction count
vangelis@vangelis-VirtualBox:~/gem5$
```

FIGURE A.2: gem5's command line output when running the bare-metal simulation.

The first section of the terminal output, shown in the figure above, contains information about gem5 along with the executed command. The second section indicates that the simulation has started and contains information about the simulation such as the global simulation frequency, listening ports and the start of execution. The last line shows the exit cause of the simulation along with the final simulation tick.

After the simulation has finished execution, gem5's output files will be inside the directory that the user has defined. If the user does not define an output directory, the output files will be stored in the folder *m5out*. In this example, output files will be inside the directory `results/benchmarks/Dhrystone/test`, as shown below.



```
vangelis@vangelis-VirtualBox:~/gem5$ ls results/benchmarks/Dhrystone/test
config.dot  config.dot.pdf  config.dot.svg  config.ini  config.json  stats.txt  system.platform.terminal  trace.out
vangelis@vangelis-VirtualBox:~/gem5$
```

FIGURE A.3: gem5's output files.

Running in Full-System mode

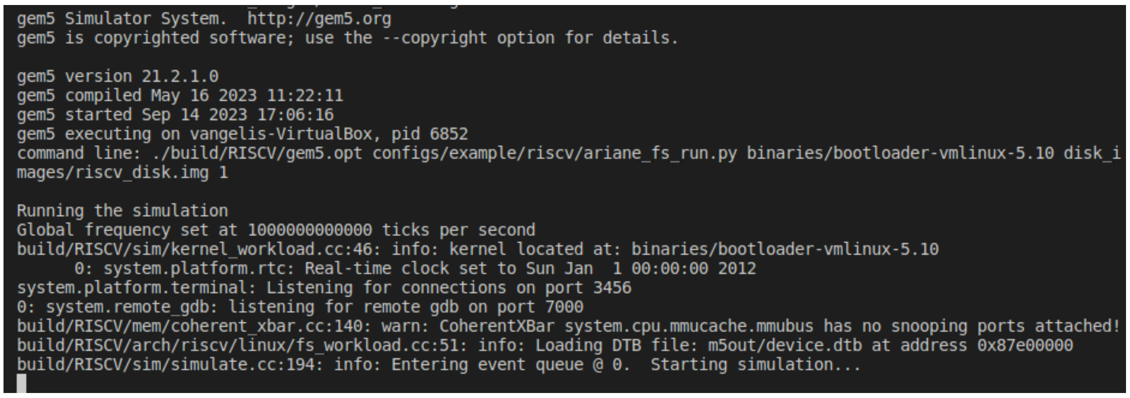
For the Full-System simulation, we need the configuration script, the kernel binary, and the disk image. The command format for running the simulation is shown below.

```
/gem5$ ./build/RISCV/gem5.opt -d [gem5 output file path] [debug flags]
[config script path] [kernel binary path] [disk image path]
[number of cores]
```

For example, to run Full-System mode using the kernel binary and disk image mentioned in section A.3, with one core, use the following command.

```
/gem5$ ./build/RISCV/gem5.opt configs/example/riscv/ariane_fs_run.py
'binaries/bootloader-vmlinux-5.10' 'disk_images/riscv_disk.img' 1
```

After executing the command, the simulation starts and produces a similar output in the command line terminal with the one shown in A.2. The FS mode terminal output, contains further information about the simulation, such as the *DTB* file location and the kernel binary used, as shown below. The system's DTB file is autogenerated by the configuration script.



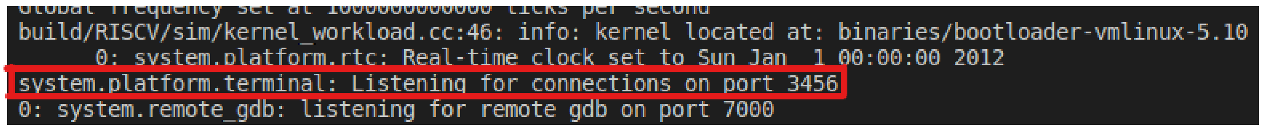
```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.2.1.0
gem5 compiled May 16 2023 11:22:11
gem5 started Sep 14 2023 17:06:16
gem5 executing on vangelis-VirtualBox, pid 6852
command line: ./build/RISCV/gem5.opt configs/example/riscv/ariane_fs_run.py binaries/bootloader-vmlinux-5.10 disk_i
mages/riscv_disk.img 1

Running the simulation
Global frequency set at 1000000000000 ticks per second
build/RISCV/sim/kernel_workload.cc:46: info: kernel located at: binaries/bootloader-vmlinux-5.10
      0: system.platform.rtc: Real-time clock set to Sun Jan  1 00:00:00 2012
system.platform.terminal: Listening for connections on port 3456
0: system.remote_gdb: listening for remote gdb on port 7000
build/RISCV/mem/coherent_xbar.cc:140: warn: CoherentXBar system.cpu.mmucache.mmubus has no snooping ports attached!
build/RISCV/arch/riscv/linux/fs_workload.cc:51: info: Loading DTB file: m5out/device.dtb at address 0x87e00000
build/RISCV/sim/simulate.cc:194: info: Entering event queue @ 0.  Starting simulation...
```

FIGURE A.4: gem5's command line output when running FS simulation.

To communicate with the simulated system, gem5 offers a serial terminal, named *m5term*. To build the terminal move to `gem5/util/term` directory and simply run `make` and then `make install`, as shown below.

```
/gem5$ cd util/term
/gem5/util/term$ make
/gem5/util/term$ make install
```

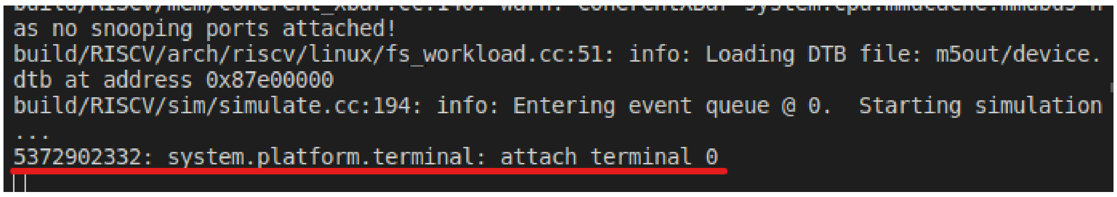
To run *m5term* use `./m5term [host] [port]`. The default port that gem5 uses for the terminal is '3456'. If this port is used, gem5 will use a higher port. The port for the terminal is indicated in the terminal when the simulation starts, as shown below. The default host is *localhost*.



```
Global frequency set at 1000000000000 ticks per second
build/RISCV/sim/kernel_workload.cc:46: info: kernel located at: binaries/bootloader-vmlinux-5.10
      0: system.platform.rtc: Real-time clock set to Sun Jan  1 00:00:00 2012
system.platform.terminal: Listening for connections on port 3456
0: system.remote_gdb: listening for remote gdb on port 7000
```

FIGURE A.5: Port indication for the serial terminal.

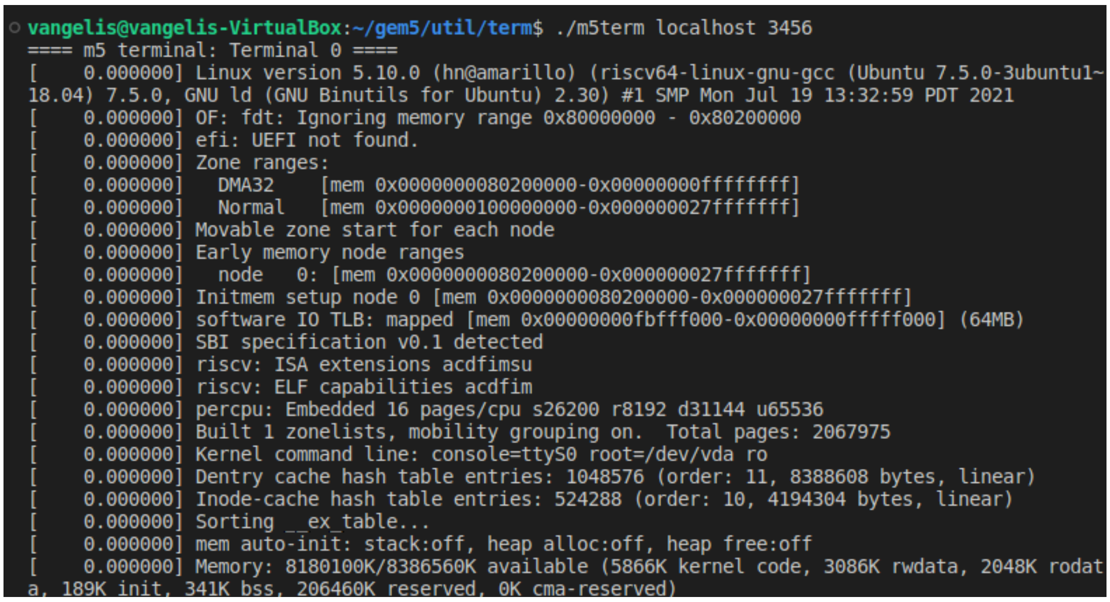
To use *m5term* in the simulation, run gem5 on FS mode with the command shown above. Then, while gem5 is running, open a new terminal window, move to directory `gem5/util/term` and run `./m5term localhost 3456`. In the terminal where gem5 is running you should see a message indicating that *Terminal 0* is connected.



```
as no snooping ports attached!
build/RISCV/arch/riscv/linux/fs_workload.cc:51: info: Loading DTB file: m5out/device.
dtb at address 0x87e00000
build/RISCV/sim/simulate.cc:194: info: Entering event queue @ 0.  Starting simulation
...
5372902332: system.platform.terminal: attach terminal 0
```

FIGURE A.6: Serial Terminal attached prompt.

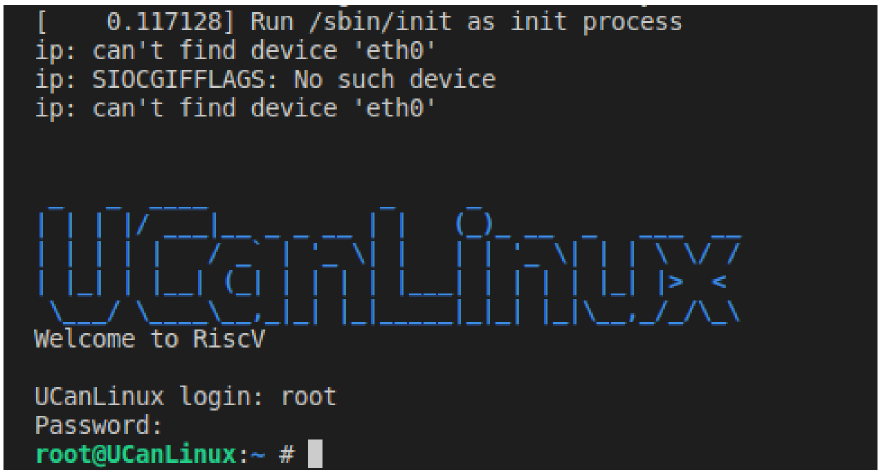
After a while, the system starts to boot and start-up information will show in the m5term's terminal window. Note that this process can take a lot of time depending on the host machine.



```
vangelis@vangelis-VirtualBox:~/gem5/util/term$ ./m5term localhost 3456
==== m5 terminal: Terminal 0 ====
[    0.000000] Linux version 5.10.0 (hn@amarillo) (riscv64-linux-gnu-gcc (Ubuntu 7.5.0-3ubuntu1~
18.04) 7.5.0, GNU ld (GNU Binutils for Ubuntu) 2.30) #1 SMP Mon Jul 19 13:32:59 PDT 2021
[    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
[    0.000000] efi: UEFI not found.
[    0.000000] Zone ranges:
[    0.000000]   DMA32    [mem 0x0000000080200000-0x00000000ffffffff]
[    0.000000]   Normal   [mem 0x0000000100000000-0x000000027fffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080200000-0x000000027fffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080200000-0x000000027fffffff]
[    0.000000] software IO TLB: mapped [mem 0x00000000fbfff000-0x00000000fffff000] (64MB)
[    0.000000] SBI specification v0.1 detected
[    0.000000] riscv: ISA extensions acdfimsu
[    0.000000] riscv: ELF capabilities acdfim
[    0.000000] percpu: Embedded 16 pages/cpu s26200 r8192 d31144 u65536
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 2067975
[    0.000000] Kernel command line: console=ttyS0 root=/dev/vda ro
[    0.000000] Dentry cache hash table entries: 1048576 (order: 11, 8388608 bytes, linear)
[    0.000000] Inode-cache hash table entries: 524288 (order: 10, 4194304 bytes, linear)
[    0.000000] Sorting __ex_table...
[    0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[    0.000000] Memory: 8180100K/8386560K available (5866K kernel code, 3086K rwdata, 2048K rodat
a, 189K init, 341K bss, 206460K reserved, 0K cma-reserved)
```

FIGURE A.7: A part of the simulated system's start-up information in m5term.

When the system boots up, a login screen will appear in m5term's terminal window, use username *root* and password *root* to log in to the system. From there you can use standard Linux commands. The *m5term* uses '~' as an escape character. Pressing '.' after pressing the escape character will exit *m5term* and terminate the command line.



```
[    0.117128] Run /sbin/init as init process
ip: can't find device 'eth0'
ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'


 _   _  ____            _     _                  
| | | |/ ___|__ _ _ __ | |   (_)_ __  _   ___  __
| | | | |   / _` | '_ \| |   | | '_ \| | | \ \/ /
| |_| | |__| (_| | | | | |___| | | | | |_| |>  < 
 \___/ \____\__,_|_| |_|_____|_|_| |_|\__,_/_/\_\
Welcome to RiscV

UCanLinux login: root
Password:
root@UCanLinux:~ #
```

FIGURE A.8: Log-in screen and command line of the simulated system.

A.5 Changing Parameter Values

You can change the basic parameters of the CPU by changing their values in the corresponding classes in script *ariane.py*.In this script, you can change the number and operation of functional units, cache parameters, branch prediction units and parameters and parameters about the CPU pipeline and pipeline components.

For example, you can change the number of ROB entries by changing the values of the respective parameters in class `Ariane()` as shown below.

```python
class Ariane(DerivO3CPU):
    ...

    numIQEntries = 16
    numROBEntries = 8

    ...
```

To change the latency of the integer divisions, change the value of `opLat` parameter in class `ArianeMultDiv()` as shown below.

```python
class ArianeMultDiv(FUDesc):
    opList = [ OpDesc(opClass='IntMult', opLat=2, pipelined=
True),
    OpDesc(opClass='IntDiv', opLat=2, pipelined=False) ]
    count = 1

    ...
```

A.6 Running McPAT

McPAT along with the helper scripts for the XML file creation used in this project can be obtained from the following link [41]. To obtain and build McPAT, type in any working directory:

```
$ git clone https://github.com/H2020-COSSIM/cMcPAT
$ cd cmcpat/mcpat
$ make
```

To create an XML input file for McPAT using gem5's output files, open a terminal inside the `cmcpat/Scripts` directory and run:

```
$ ./GEM5ToMcPAT.py [options] <gem5 stats file> <gem5 config file (json)>
<mcpat template xml file> -o <output xml file>
```

You have to enter the paths of the *stats.txt* and *config.json* files that gem5 produces from a simulation, along with the path of the provided template, in the corresponding fields. In the *output xml file* enter the path where the final XML file will be stored. After the XML file is created, it is advised that you copy and move it to the *ProcessorDescriptionFiles* folder inside the `/cmcpat/mcpat` directory.

To run McPAT, open a terminal inside the `/cmcpat/mcpat` directory and then run:

```
$ ./mcpat -infile ProcessorDescriptionFiles/<your XML file> -print_level 5
```

Then McPAT will print the results in the command line terminal. Option `-print_level` defines the detail of the results that McPAT will print. The maximum detail level is 5 and the lowest is 1. To save the results you can redirect the terminal's output to a file. For example:

```
$ ./mcpat -infile ProcessorDescriptionFiles/test.xml -print_level 5
> test_out.txt
```

Folder *ProcessorDescriptionFiles* contains a number of example XML files for different configurations.

Appendix B

Common Errors during setup and development with gem5 and possible solutions

When working with gem5 and the RISC-V toolchain many errors may occur during setup and development. In this appendix, we present common errors that may occur along with a possible solution.

B.1 Errors on compilation

1. When building gem5, make sure that you have all the prerequisite packages already installed before setup. If the installation still fails after obtaining all the required packages try a clean build by using `scons --clean` and then rebuild gem5 (or delete the *build* directory in gem5's main directory and rebuild gem5).

2. When building *m5ops* for the RISC-V ISA, make sure that the RISC-V toolchain is installed and the `--enable-multilib` flag is enabled when building the toolchain. If not, rebuild the toolchain with the *multilib* flag. Appendix A provides information on the configuration of the RISC-V compiler toolchain.

3. When building the *riscv-tests* suite or any binary using the cross-compilers from RISC-V toolchain, make sure that the RISC-V Linux variable, `$RISCV` is set and then add it to `$PATH`. Every time you use a new terminal you have to set the variable and add it to the path. More information is provided in Appendix A.

4. If using *m5ops* in your code, first of all, make sure that you have already built the library *libm5.a* and that you have added *gem5/include* to the compiler include path and *gem5/util/m5/build/RISCV/out* to the linker path. Library *gem5/m5ops.h* needs to be included in the source file. More information on m5ops is provided in [53].

B.2 Errors during development with gem5's Full-System mode

1. If an *illegal instruction* error occurs during the simulation, it is probably caused because the simulated binary runs on a privileged level not supported by the simulation mode. Syscall emulation mode only supports user-level binaries, try switching between FS and bare-metal mode.
2. If the simulation crashes due to memory and I/O addresses overlapping, set the *bad_addr responder* parameter, or use the *MemBus()* bus instead of the *SystemXBar* bus.
3. The simulation often produces a warning stating that the main memory size must be over *8GB*. It does not really affect the simulation, however, you can set a memory size of *8GB* to fix it.
4. When running in FS mode with the serial terminal connected, make sure that you connect the terminal to the correct port. The serial terminal port number is indicated in gem5's terminal output when the simulation begins. During the simulation, when the terminal is connected, gem5 produces a warning stating that an address is *outside of physical memory*. It is a silent warning and doesn't seem to affect the simulation.
5. In bare-metal mode, when running tests from the *riscv-tests* suite make sure to set the instruction limit, otherwise the simulation will never stop executing. To use the instruction limit, make sure that you use the *Run()* method instead of *m5.simulate()* in your configuration script.
6. When running Full-System simulation, with your own scripts, the simulation may crash after the OS boots up with a *snoop filter* error. To fix the issue we set the *mmucache* module in the configuration script.

B.3 Errors when using McPAT

1. If the script that converts gem5's output to McPAT's input produces an error stating that *stat or param is missing*, check gem5's output files and make sure that the name of the statistic or parameter is the same with the one on the template.

2. McPAT may produce a *no valid data array organization*. This is probably caused since McPAT cannot create a possible circuit model for the given configuration, it is especially common when working with smaller cores. In our case, increasing the size of the reorder buffer seems to fix the issue.

References

[2] Florian Zaruba and Luca Benini. “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. DOI: 10.1109/TVLSI.2019.2926114.

[3] Krste Asanović and David A. Patterson. “Instruction Sets Should Be Free: The Case For RISC-V”. In: UCB/EECS-2014-146 (Aug. 2014). URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[5] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Tech. rep. EECS Department, University of California, Berkeley, May 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[6] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: ATEC ’05 (2005), p. 41.

[7] Matheus Cavalcante et al. “Ara: A 1-GHz Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.2 (Feb. 2020), pp. 530–543. DOI: 10.1109/tvlsi.2019.2950087. URL: https://doi.org/10.1109%2Ftvlsi.2019.2950087.

[9] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[10] Jonathan Bachrach et al. “Chisel: Constructing Hardware in a Scala Embedded Language”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. San Francisco, California: Association for

Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. URL: https://doi.org/10.1145/2228360.2228584.

[11] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, June 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[12] Christopher Celio et al. *BOOM v2: an open-source out-of-order RISC-V core*. Tech. rep. UCB/EECS-2017-157. EECS Department, University of California, Berkeley, Sept. 2017. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

[13] Christopher Celio et al. "BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS". In: *IEEE Micro* 39.2 (2019), pp. 52–60. DOI: 10.1109/MM.2019.2897782.

[14] Jerry Zhao et al. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: (May 2020).

[15] Reinhold Weicker. "Dhrystone: a synthetic systems programming benchmark". In: *Commun. ACM* 27 (1984), pp. 1013–1030. URL: https://api.semanticscholar.org/CorpusID:9026014.

[16] Alan R Weiss. "Dhrystone benchmark". In: *History, Analysis,„Scores "and Recommendations, White Paper, ECL/LLC* (2002).

[17] Richard York. "Benchmarking in context: Dhrystone". In: *ARM, March* (2002).

[18] Reinhold Weicker. "Dhrystone benchmark: rationale for version 2 and measurement rules". In: *ACM SIGPLAN Notices* 23 (1988), pp. 49–62. URL: https://api.semanticscholar.org/CorpusID:22774415.

[19] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[20] Jason Lowe-Power et al. "The gem5 Simulator: Version 20.0+". In: (2020). arXiv: 2007.03152 [cs.AR].

[21] N.L. Binkert et al. "The M5 Simulator: Modeling Networked Systems". In: *IEEE Micro* 26.4 (2006), pp. 52–60. DOI: 10.1109/MM.2006.82.

[22] Milo M. K. Martin et al. "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset". In: *SIGARCH Comput. Archit.*

News 33.4 (Nov. 2005), pp. 92–99. ISSN: 0163-5964. DOI: 10.1145/1105734.1105747. URL: https://doi.org/10.1145/1105734.1105747.

[23] Sheng Li et al. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: (2009), pp. 469–480.

[24] Sam Likun Xi et al. "Quantifying sources of error in McPAT and potential impacts on architectural studies". In: (2015), pp. 577–589. DOI: 10.1109/HPCA.2015.7056064.

[25] Wooseok Lee et al. "PowerTrain: A learning-based calibration of McPAT power models". In: (2015), pp. 189–194. DOI: 10.1109/ISLPED.2015.7273512.

[26] Jianwang Zhai et al. "McPAT-Calib: A RISC-V BOOM Microarchitecture Power Modeling Framework". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.1 (2023), pp. 243–256. DOI: 10.1109/TCAD.2022.3169464.

[27] Aoxiang Tang et al. "McPAT-PVT: Delay and Power Modeling Framework for FinFET Processor Architectures Under PVT Variations". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.9 (2015), pp. 1616–1627. DOI: 10.1109/TVLSI.2014.2352354.

[28] Jianwang Zhai et al. "McPAT-Calib: A Microarchitecture Power Modeling Framework for Modern CPUs". In: (2021), pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643508.

[29] Abdullah Guler and Niraj K. Jha. "McPAT-Monolithic: An Area/Power/Timing Architecture Modeling Framework for 3-D Hybrid Monolithic Multicore Systems". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.10 (2020), pp. 2146–2156. DOI: 10.1109/TVLSI.2020.3002723.

[30] Anastasiia Butko et al. "Accuracy evaluation of GEM5 simulator system". In: (2012), pp. 1–7. DOI: 10.1109/ReCoSoC.2012.6322869.

[31] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. "Microarchitectural simulation of in-order and out-of-order ARM microprocessors with gem5". In: (2014), pp. 266–273. DOI: 10.1109/SAMOS.2014.6893220.

[32] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. "Micro-Architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT". In: RAPIDO '15 (2015). DOI: 10.1145/2693433.2693440. URL: https://doi.org/10.1145/2693433.2693440.

[33] Anastasiia Butko et al. "Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration". In: (2016), pp. 201–208. DOI: 10.1109/MCSoC.2016.20.

[34] Anas Ahmad Abudaqa et al. "Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator". In: (2018), pp. 317–322. DOI: 10.1109/ICEEE2.2018.8391354.

[35] Mircea R. Stan Alec Roelke. "RISC5: Implementing the RISC-V ISA in gem5". In: (2017). URL: https://api.semanticscholar.org/CorpusID:221881571.

[36] Tuan Ta, Lin Cheng, and Christopher Batten. "Simulating multi-core RISC-V systems in gem5". In: (2018). URL: https://www.csl.cornell.edu/~cbatten/pdfs/ta-gem5-riscv-carrv2018.pdf.

[37] Peter Yuen Ho Hin et al. "Supporting RISC-V full system simulation in gem5". In: (2021). URL: https://carrv.github.io/2021/papers/CARRV2021_paper_7_Yuen.pdf.

[38] Odysseas Chatzopoulos et al. "Towards Accurate Performance Modeling of RISC-V Designs". In: (2021). arXiv: 2106.09991 [cs.AR].

[39] Renan Tashiro and Marcio Seiji Oyamada. "An Environment for Design Space Exploration Using gem5-McPAT". In: (2016), pp. 220–225. DOI: 10.1109/SBESC.2016.042.

[42] Nikolaos Tampouratzis et al. "A Novel, Highly Integrated Simulator for Parallel and Distributed Systems". In: *ACM Trans. Archit. Code Optim.* 17.1 (Mar. 2020). ISSN: 1544-3566. DOI: 10.1145/3378934. URL: https://doi.org/10.1145/3378934.

External Links

[1] "Spike RISC-V ISA Simulator". In: (). URL: https://github.com/riscv-software-src/riscv-isa-sim.

[8] "RISC-V cores and SoC Overview". In: (). URL: https://github.com/riscvarchive/riscv-cores-list.

[40] "The gem5 Simulator". In: (). URL: https://www.gem5.org/.

[41] "cMcPAT". In: (). URL: https://github.com/H2020-COSSIM/cMcPAT.

[43] "RISC-V bootlader with Linux kernel v5.10". In: (). URL: http://dist.gem5.org/dist/v22-0/kernels/riscv/static/bootloader-vmlinux-5.10.

[44] "Simple RISC-V disk image based on busybox". In: (). URL: http://dist.gem5.org/dist/v22-0/images/riscv/busybox/riscv-disk.img.gz.

[45] "Documentation on gem5-resources from gem5.org". In: (). URL: https://www.gem5.org/documentation/general_docs/gem5_resources/.

[46] "Building gem5". In: (). URL: https://www.gem5.org/documentation/general_docs/building.

[47] "The RISC-V GNU Compiler toolchain". In: (). URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[48] "RISC-V tests repository". In: (). URL: https://github.com/riscv-software-src/riscv-tests.

[49] "gem5-resources repository". In: (). URL: https://github.com/gem5/gem5-resources.

[50] "gem5 Resources website". In: (). URL: https://resources.gem5.org/.

[51] "RISC-V Full-System guide". In: (). URL: https://github.com/gem5/gem5-resources/tree/develop/src/riscv-fs.

[52] "Creating and mounting disk images". In: (). URL: https://www.gem5.org/documentation/general_docs/fullsystem/disks.

[53] "M5ops". In: (). URL: https://www.gem5.org/documentation/general_docs/m5ops/.