**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

# Αξιολόγηση της Ακρίβειας και των Επιδόσεων Μοντέλων Επεξεργαστών σε Επίπεδο Μεταφοράς Καταχωρητών και Μικροαρχιτεκτονικό Επίπεδο

**Γεώργιος-Μάριος Κ. Φραγκούλης**
**Οδυσσέας Δ. Χατζόπουλος**

**ΕΠΙΒΛΕΠΩΝ: Δημήτριος Γκιζόπουλος**, Καθηγητής

**ΑΘΗΝΑ**

**ΣΕΠΤΕΜΒΡΙΟΣ 2021**

**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS**

**SCHOOL OF SCIENCES**
**DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

**BSc THESIS**

# Evaluation of the Accuracy and the Performance of Register Transfer Level and Microarchitecture Level CPU Models

**Odysseas D. Chatzopoulos**
**George-Marios K. Fragkoulis**

**SUPERVISOR: Dimitris Gizopoulos**, Professor

**ATHENS**

**SEPTEMBER 2021**

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

# Αξιολόγηση της Ακρίβειας και των Επιδόσεων Μοντέλων Επεξεργαστών σε Επίπεδο Μεταφοράς Καταχωρητών και Μικροαρχιτεκτονικό Επίπεδο

**Γεώργιος-Μάριος Κ. Φραγκούλης**
**Α.Μ.:** 1115201700179

**Οδυσσέας Δ. Χατζόπουλος**
**Α.Μ.:** 1115201700191

**ΕΠΙΒΛΕΠΩΝ: Δημήτριος Γκιζόπουλος**, Καθηγητής

**BSc THESIS**

# Evaluation of the Accuracy and the Performance of Register Transfer Level and Microarchitecture Level CPU Models

**Odysseas D. Chatzopoulos**
**S.N.:** 1115201700191

**George-Marios K. Fragkoulis**
**S.N.:** 1115201700179

**SUPERVISOR: Dimitris Gizopoulos**, Professor

## ΠΕΡΙΛΗΨΗ

Στην συγκεκριμένη εργασία συγκρίνουμε την RTL προσομοίωση με την προσομοίωση σε μικροαρχιτεκτονικό επίπεδο, επισημαίνοντας το ισοζύγιο απόδοσης και ακρίβειας μεταξύ αυτών των δύο. Στην προσπάθειά μας να εκμεταλλευτούμε την ταχύτητα και την προσαρμοστικότητα του μικροαρχιτεκτονικού επιπέδου, χωρίς να παραμερίζουμε την επίπτωση στην ακρίβεια, διαμορφώνουμε το μοντέλο μικροαρχιτεκτονικού επιπέδου σε σχέση με το μοντέλο RTL. Η μελέτη μας επικεντρώθηκε σε έναν υπερβαθμωτό, εκτός σειράς πυρήνα, ο οποίος χρησιμοποιεί RISC-V αρχιτεκτονική συνόλου εντολών. Η προσπάθεια αντιστοίχισης των δύο μοντέλων, η οποία επιτεύχθηκε μέσω έρευσης της τιμής συμαντικών μικροαρχιτεκτονικών παραμέτρων και εκτελέσεων στοχευμένων προγραμμάτων, οδήγησε σε σφάλμα προσομοίωσης 15.35%. Κλέινοντας, στο μέλλον σχεδιάζουμε να ανακάλυψουμε επιπλέον τιμές αρχιτεκτονικών παραμέτρων και να βελτιώσουμε την ακρίβεια της προσομοίωσης μας χρησιμοποιώντας εξελιγμένους επεξεργαστές και full system μοντέλα.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ**: Αρχιτεκτονική Υπολογιστών

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ**: αρχιτεκτονική υπολογιστών, προσομοίωση RTL, προσομοίωση σε μικροαρχιτεκτονικό επίπεδο, επικύρωση μοντέλου, RSD, gem5

## ABSTRACT

In this work we compare RTL with microarchitecture-level simulation highlighting the performance vs accuracy trade-off between the two. In an effort to benefit from the higher speeds and flexibility of microarchitecture-level simulation while not significantly affecting simulation accuracy we strive to fine-tune the microarchitectural model to closely match the RTL one. Throughout this work we make use of the RISC-V ISA targeting a superscalar out-of-order open-source core. After going through our matching process which includes microarchitectural parameter discovery and matching followed by thorough benchmarking we achieve a 15.35% simulation error. In future work we plan to streamline the microarchitectural parameter discovery and improve simulation accuracy while also use more advanced processor and full system models.

**SUBJECT AREA**: Computer Architecture

**KEYWORDS**: computer architecture, RTL simulation, microarchitecture-level simulation, model validation, RSD, gem5

## ACKNOWLEDGEMENTS

We would like to thank our families with all our hearts since they have supported us un-codintionally for the longest amount of time making personal sacrifices so that we could pursue our dreams.

We would also like to extend our deepest gratitude to our thesis advisor Professor Dimitris Gizopoulos for his continued guidance and invaluable support.

Finally we would like to convey our thanks to Professor Ryota Shioya at the University of Tokyo for his availability and promptness to help with matters concerning RSD.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1. INTRODUCTION

## 1.1 Computer Architecture Primer

In the last few decades the field of computing has seen major improvement and has transformed the way that people consume their entertainment, communicate with each other and work. Aside from affecting the daily lives of billions of people, computers are critical in solving some of the most important problems humanity has to face such as finding novel ways to fight disease, predicting and controlling the effects of climate change and understanding the hidden secrets of the universe. One of the main drivers behind the rapid growth of computing is the continued improvement of computer hardware that enables more complex and sophisticated software to be developed, tested and executed.

Computer architecture is the field of computer science and engineering that encompasses the design of computer systems balancing performance and energy efficiency according to expected user demands while staying within cost, power, area and reliability constraints. A computer architect must be well versed in the design of the instruction set architecture (ISA), computer organization (microarchitecture), digital design and physical implementation topics such as integrated circuit design, packaging and power management techniques [28], [35]. Nowadays computer systems are very complex and contain a mix of microprocessors (central processing units - CPUs), graphics processing units (GPUs), domain specific accelerators and high speed peripherals often in one system on a chip (SoC). Computer architects are expected to combine these components to create a computer system that meets the user's requirements in terms of power, performance and reliability. The microprocessor (CPU) is the central component of such system. It executes the bulk of program code while also coordinating the rest of the system components.

When designing a microprocessor the architect must first design a new instruction set architecture (ISA) or utilize an existing one. The ISA is the interface between the hardware and software layers of a computer system. It specifies the behavior of machine code instructions. According to [28], there are seven main aspects of an ISA.

i. *Class of ISA*. The vast majority of ISAs used nowadays are general-purpose register architectures. They use registers and memory addresses as operands for their instructions. Most modern ISAs belong to a subgroup of the aforementioned class called *Load-Store* ISAs where memory operands are only used in load and store instructions and not in arithmetic or logic operations.

ii. *Memory addressing*. Almost all recent desktop and server class ISAs employ byte addressing to access the memory system. Some of them require that memory objects be aligned[1] whereas others allow unaligned accesses usually at a reduced speed due to multiple memory accesses required for unaligned operands.

[1]The requirement that the address of an object is always a multiple of its size in bytes

iii. *Addressing modes*. Addressing modes determine the way that operands are defined in machine code instructions. This includes registers, constants and memory addresses.

iv. *Types and sizes of operands*. Most ISAs support a range of operand sizes. Common operand sizes are 8-bits (ASCII character), 16-bits (Unicode character or half word), 32-bits (integer, single precision floating point or word) and 64-bits (long integer, double precision floating point or double word).

v. *Operations*. ISAs usually include a variety of arithmetic, logical, data transfer, control and floating point operations. More complex and special-purpose instructions are often included to accelerate certain workloads.

vi. *Control flow instructions*. Effectively all ISAs support control flow instructions like conditional and unconditional branches, routine calls and returns. For such instructions PC-relative addressing is most commonly used.

vii. *Encoding an ISA*. There are two main approaches to encoding instructions. Fixed-length and variable-length encoding. The main trade-off between these two is code size and complexity of the decode hardware. Using fixed-length encoding can reduce the size of the decode unit but usually results in larger compiled code size.

After deciding on an ISA, the architect must begin implementing it. Nowadays, with ISAs being quite similar and new ISA design rare, the bulk of the design effort is put into the implementation [28] to maximize performance or other parameters. There are two main parts that go into implementing an ISA, *organization* and *hardware implementation*. Organization or otherwise known as microarchitecture refers to the high-level design of the memory system, the memory interconnect and the microprocessor core where the functional and control units of the microprocessor reside. The hardware implementation includes the complete logic design, integrated circuit design, fabrication and packaging of the microprocessor or SoC.

Figure 1.1 depicts the abstraction layers of a computer system that computer architects mostly deal with. The two middle layers i.e the ISA and the Microarchitecture are the main focus of computer architecture but the two layers above and the two layers below them are very important to keep in mind to achieve optimal performance and stay within power, cost and area constraints.

## 1.2 A Brief History of Microprocessors

The term *microprocessor* refers to a computer processor where the datapath and control digital logic are included in a single integrated circuit. The microprocessor accepts binary data as input and processes it according to stored instructions. The integration of the entire processor into a single chip reduces power consumption and cost and contributes to increased performance when compared to a discrete processor design. Microprocessors

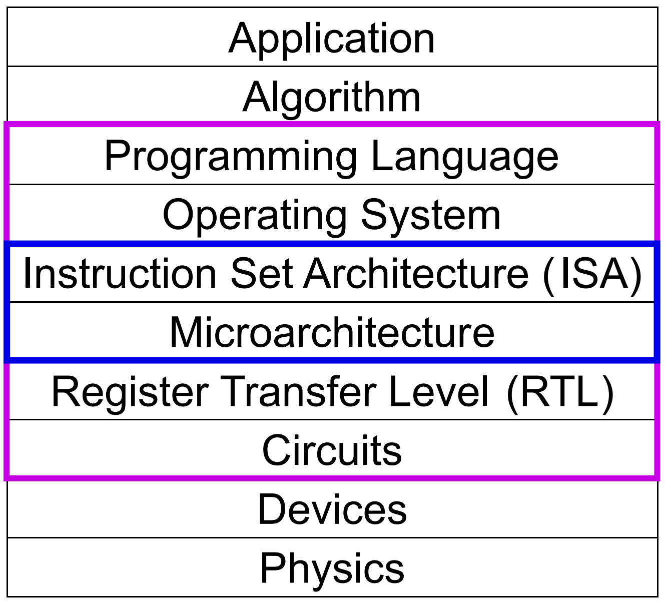


**Figure 1.1: Computer Architect view of the Computer System Stack**

are also more reliable than their discrete counterparts. The Intel 4004 is considered to be the first commercial microprocessor design, having used 2300 integrated transistors in 10um technology, being released in 1971 [24]. From that point of time onwards microprocessors have become increasingly complex in order to continually provide better performance and power efficiency. There are two main driving forces behind this generational improvement: advancement of semiconductor technology and architectural and microarchitectural innovation.

Figure 1.2 plots performance of microprocessors over time. Yearly improvement varies between time periods. During the first years of microprocessors, performance increased at a steady pace of 25% every year. This growth was mainly driven by improvements in semiconductor technology. After 1986 performance improvement jumped to 52% yearly which is mainly ascribed to more advanced architectures and microarchitectures associated with RISC designs. After 2003 which signified the end of Dennard scaling[2] and the available instruction-level parallelism (ILP), improvements dropped back again to 23% annually. In the 2011 to 2015 period performance improvement dropped again to 12% mainly due to the inherent limitations of parallel computing as described by Amdahl's law[3]. Finally, from 2015 to 2018, with the end of Moore's law[4] improvement was just 3.5% per year. In the last few years though, due to a more aggressive push towards newer manufacturing processes and microarchitectures from companies like AMD and Apple, generational performance improvements have somewhat increased as shown in Figures 1.3, 1.4.

To achieve maximum performance in the post Moore's law era microarchitectures usually

[2]Scaling law stating that as transistors get smaller their power density stays constant making the power use proportional to the circuit area

[3]A formula which gives the maximum theoretical speedup of a fixed workload upon the improvement of one part of the system that executes it

[4]The observation that roughly every two years the number of transistors in dense ICs doubles

implement the following common design features [25].

- *Pipelining*. Pipelining divides the execution of instructions into different stages allowing several instructions in different phases to be processed simultaneously (partial overlapping of instructions execution). This technique increases instruction level parallelism. Virtually all mobile, desktop and server class microprocessors nowadays implement pipelining due to it's inexpensive implementation and significant contribution to performance.

- *Out-of-Order Execution*. Out-of-Order (OoO) microprocessors don't necessarily execute instructions in program order but execute them in an order based upon the availability of the instruction operands and the execution units. OoO execution or otherwise known as Dynamic Scheduling increases the amount of instruction level parallelism by reducing avoidable stall cycles.

- *Superscalar Execution*. Superscalar microprocessors can execute more than 1 instruction simultaneously in all pipeline stages (full overlapping of instructions execution). This means that a throughput higher than 1 instruction per cycle can be achieved (depending on the code being executed).

- *Vector Extensions*. Most microprocessors nowadays support vector instructions. These instructions are also known as SIMD instructions (Single Instruction Multiple Data). Vector instructions increase data level parallelism.

- *Multiple Cores*. Microprocessors nowadays have more than one core. A core is a standalone entity that can execute a stream of instructions called a thread. Multicore

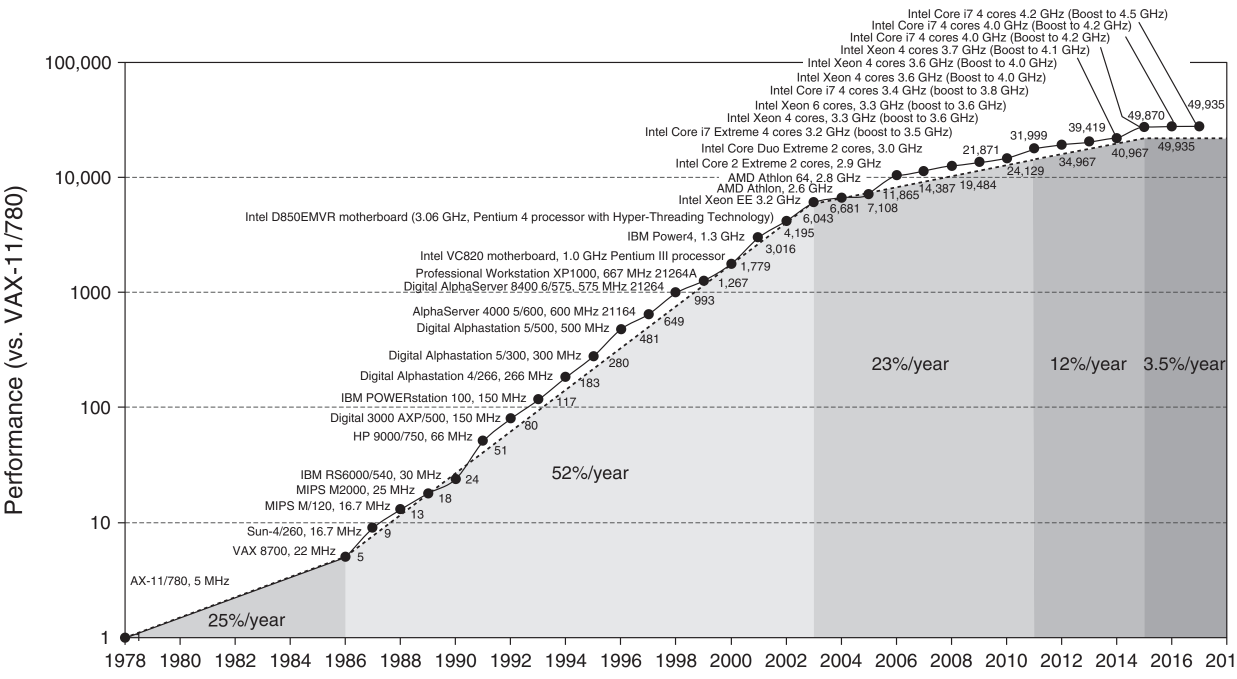


**Figure 1.2: Processor Performance Evolution [28]**

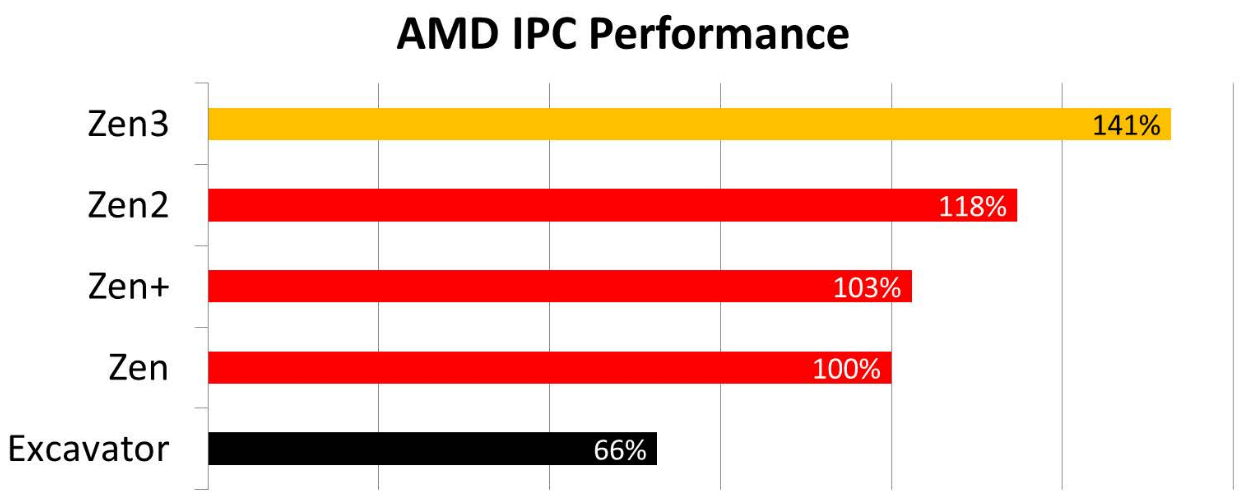


**Figure 1.3: AMD Zen Architecture IPC Improvements [21]**

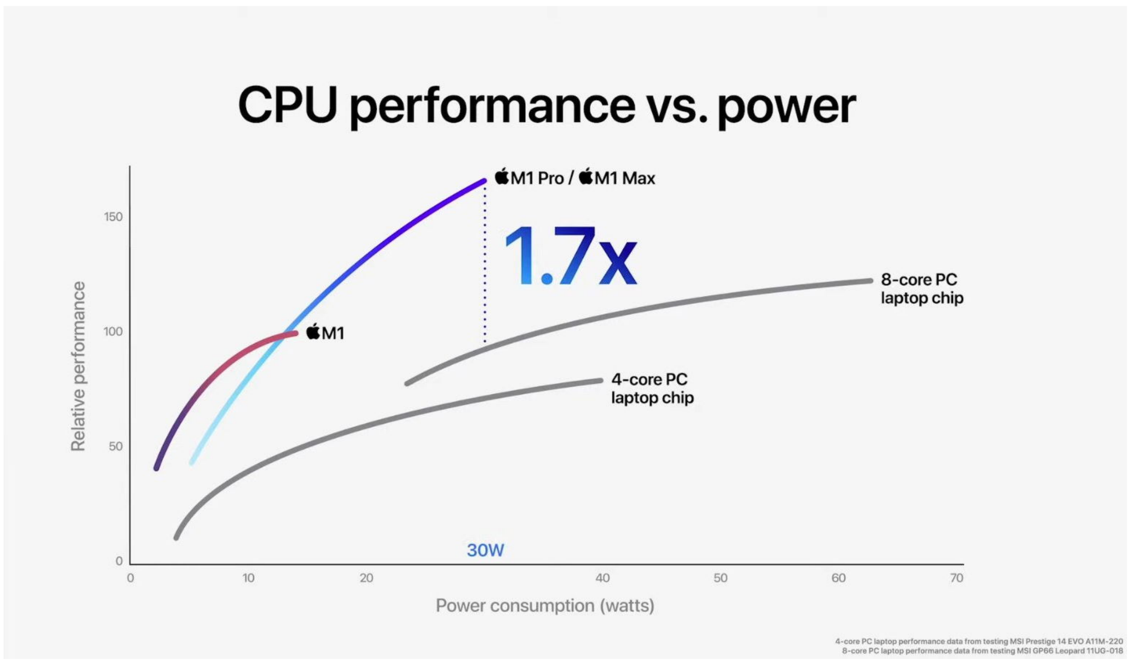


**Figure 1.4: Apple M1 Performance vs Power [1]**

processors can execute multiple threads simultaneously using separate hardware resources. Synchronization and communication among threads is provided.

- *Multithreading*. Multithreading allows a single core of a microprocessor to execute multiple instructions simultaneously. The key difference between multicore and multithreaded processors is that simultaneously executed instructions use separate hardware in multicore processors whereas in multithreaded processors they utilize mostly the same hardware. Multithreading is often implemented for each core in a multicore microprocessor.

As mentioned above, design complexity has increased exponentially since the advent of the first microprocessor 50 years ago. Nowadays, some top of the line chips contain almost 40 billion transistors in 7nm technology. This increase in complexity requires a layered approach to design in order to create microprocessors within reasonable time frames that not only are high performance but also present high reliability. One crucial part of this multi-layered design effort is simulation. Simulation is widely used for design-space exploration, debugging and power and performance estimation. Without simulation frameworks companies would have to spend millions in hardware prototypes that would be shortly discarded. Aside from cost the time to design microprocessors would be highly affected since foundries that manufacture chips using cutting edge fabrication nodes are few and have extremely tight schedules especially for low volume orders. In the next section we will discuss about different levels of microprocessor simulation and present the key differences between them.

## 1.3 Microprocessor Simulation Levels

Simulating a microprocessor is a multifaceted endeavour and thus there are many different approaches that suit different use cases. There exist many different simulation levels that differ in simulation speed, accuracy, degree of customization and level of abstraction. The four main simulation levels that are widely used in both academia and industry are *Application Binary Interface Simulation*, *Instruction Set Architecture Simulation*, *Microarchitecture Simulation* and *Register Transfer Level Simulation* [30].

Application Binary Interface (ABI) simulation is at the highest level of abstraction. The ABI specifies an interface for program interaction, usually between a user program and a library or operating system. The specification determines procedure call conventions, data types and system calls. ABI simulation entails implementing an ABI of one system on another system using the latter system's primitive constructs. An example of such simulation is the Windows Subsystem for Linux (Version 1) [30].

Instruction Set Architecture (ISA) simulation is one step below ABI simulation in the abstraction ladder. At this level the result of executed instructions is simulated without taking into account the digital logic that a real processor implementation entails including the notion of power and timing [30]. This type of simulation is also known as functional simulation and can be found in various commercial and open-source projects such as Vmware Workstation, QEMU and Spike. When using this simulation method the user can run binaries that are compiled for the target system without modification.

When in need of a more detailed simulation platform (i.e one that more faithfully resembles the hardware) the next logical step is Microarchitecture Simulation. Microarchitectural simulators use software to emulate the different microprocessor hardware components such as the fetch unit, the decode unit, the execution units, the memory system and scheduling block. Such simulators are often used at some point of the design of a microprocessor to perform design-space exploration. Some examples of microarchitectural simulators are Simics [32], PTLsim [44], SimpleScalar [15], OVPsim [4], Spike [6], MARSSx86 [3], Sniper [7] and gem5 [2] [18].

Register Transfer Level (RTL) simulation is the most detailed simulation one can run while still being able to practically simulate an entire microprocessor (running gate level or transistor level simulations is highly impractical for such large scale designs). RTL models directly describe hardware using a hardware description language like (System)Verilog or VHDL. These models are inherently cycle-accurate since virtually the same RTL code is used to synthesize the actual hardware. RTL simulation can be run on different stages of the design process namely before synthesis which is called *behavioral RTL simulation* or after synthesis and technology mapping which is called *post-implementation RTL simulation*. The latter does not only provide the cycle count and verification of functional correctness but also includes timing, power and area data [39].

Microarchitecture and RTL simulation are the most useful in microprocessor design. The

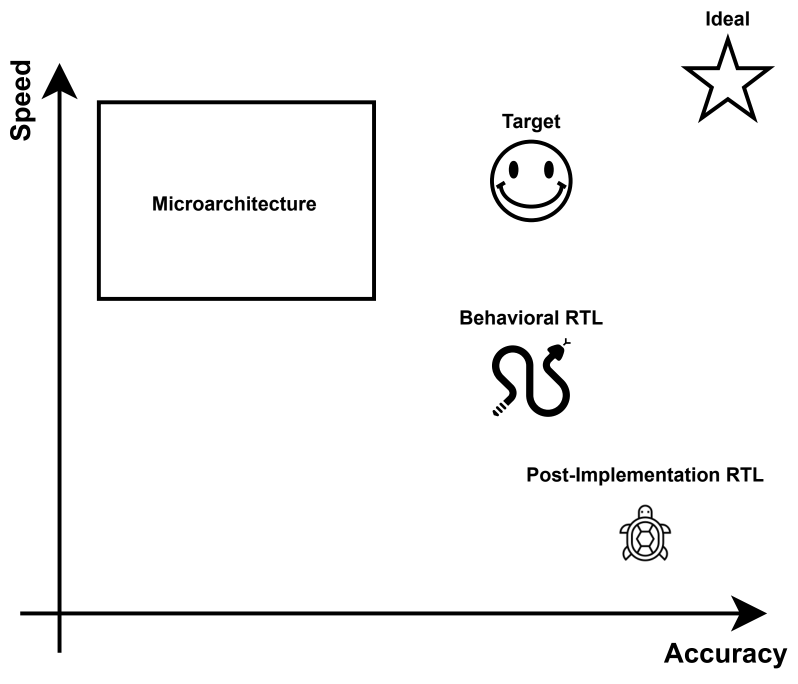


**Figure 1.5: Simulation Speed vs. Accuracy**

main trade-off between the two is accuracy versus speed as can be seen in Figure 1.5. Microarchitecture simulation, depending on the level of detail of the software emulating the microprocessor components, can be moderately to very fast whereas RTL simulation even when it is just behavioral is relatively slow. The advantage of RTL simulation is accuracy of the hardware modeling, making cycle-accurate simulation almost always guaranteed. Moving to a post-implementation model also provides us with timing, power and area information. One key advantage of microarchitectural simulation is that it is a powerful design-space exploration tool as software models are easy to modify and there exists a large library of off-the-shelf components to choose from. In recent years significant efforts have been made to design configurable RTL models and also create libraries of hardware components that can be used interchangeably [16], [12]. In our opinion such projects along with hardware accelerated RTL simulation may be the future of hardware design yet the maturity of the provided frameworks is currently not on par with widely used microarchitectural simulators.

## 1.4 Thesis Goal

In this work we focus on the tradeoffs between cycle-accurate microarchitecture simulation vs. cycle-accurate behavioral RTL simulation for performance analysis. Cycle-accurate models utilize detailed component descriptions with many parameters thus creating hazards for simulation error to occur when comparing two such models. These errors can occur due to parameter mismatch or inherent differences in the component implementation.

We aim to fine-tune the parameters of a *fast* microarchitectural model in a widely used microarchitectural simulator to match a *detailed* behavioral RTL microprocessor model. We intend to fully benefit from the higher throughput, easily modifiable and mature microarchitecture simulation framework while minimizing the simulation error as much as possible, leveraging our knowledge of key hardware details of the RTL model. The key metric of simulation accuracy used is the cycles ratio of the two models. In cases of high simulation discrepancy the main sources of error will be explored.

After an extensive literature review we have identified very few publications that tried to achieve similar goals [10], [23], [19], [18]. What differentiates our work from previous efforts is the use of a RISC-V Out-of-Order RTL microprocessor model which to the best

**Table 1.1: Simulation Throughput in Hz**

| Simulation Levels | Throughput Rate in Hz |
|---|---|
| Application Binary Interface | $\approx 10^9$ |
| Instruction Set Architecture | $\approx 10^6$ |
| Microarchitecture | $\approx 10^5$ |
| Register Transfer Level | $\approx 10^3$ |

of the authors' knowledge has not been attempted before. Most studies focus on ARM and x86 ISAs whereas RISC-V based works use In-Order cores and FPGA based simulation.

## 1.5 RISC-V Instruction Set Architecture

RISC-V is a state-of-the-art, license-free, royalty-free, open-standard ISA specification that was originally designed at UC Berkeley [5]. Nowadays RISC-V is maintained by RISC-V International a non-profit organization based in Switzerland to avoid conflicts due to US trade regulations. Before delving into the basics of RISC-V one needs to understand why an open ISA specification is needed.

According to [14] there are no good technical reasons why current popular ISAs are proprietary. The decision to make an ISA proprietary is usually profit driven. Companies patent specific peculiarities of their designs and then sell licenses that can cost somewhere in the range of $1M - $10M and can even restrict implementation of the ISA to a few company-created and approved designs. This decision is certainly sound in a business sense but contributes to the creation of monopolies and the suppression of competition and innovation. Furthermore, the notion that only big companies can design an ISA is incorrect, nor are commercial ISAs pinnacles of elegant and streamlined design as they often include obscure instructions for backwards compatibility and patent purposes. Finally, the free and open-source distribution of a processor designed using a licensed proprietary ISA is impossible since modification or use of the design in a commercial product would result in a patent violation. It is thus clear that a free open-source ISA would greatly benefit the industry as it would facilitate a truly open market of processor designs. As observed in the software world this would enable innovation, decrease the cost of SoCs and provide businesses and researchers with open processor designs which they can customize and use for their own purposes.

Since the need for an open ISA is now apparent we will explore why RISC-V is well suited to fit the role of the leading open ISA.

- RISC-V as the name implies is a Reduced Instruction Set Computer (RISC) ISA. The number five refers to the number of generations of RISC architecture that were developed at the University of California, Berkeley. All commercial ISAs nowadays are RISC ISAs or translate their instructions to RISC microoperations [40]. The benefits of RISC ISAs have been extensively discussed in Computer Architecture books and publications but the aforementioned fact should convince even the most skeptical readers.

- RISC-V is designed to support all classes of computers, from the tiniest Internet of Things (IoT) devices to the largest Warehouse-Scale Computers (WSCs). This is achieved by meeting four important requirements [14].

    i. Be a *Base + Extension ISA*. This allows the ISA to have a small core set of instructions that compilers and operating systems can use while also providing standard extensions such as floating point or bit operations for specific applications. There is also space for custom instructions to control application-specific accelerators present in the SoC. Application specific accelerators are already widely used in practice and are according to prominent computer architects a way to counter the slowing of Moore's Law [29].

ii. *Compact Code Size*. This allows area and cost constrained devices such as embedded microcontrollers to utilize their limited memory more efficiently.

iii. *Quadruple Precision Floating Point Support* alongside Single and Double Precision Floating Point. Large simulations and scientific computations that run on WSCs require extreme floating point accuracy and support for very large numbers.

iv. *128-bit Addressing alongside 32-bit and 64-bit Addressing*. This ensures that computers from the tiniest IoT devices to the largest WSCs will have the appropriate address size even after many decades have passed.

- RISC-V has seen wide community adoption and is being used extensively in industry and academia. This momentum has in our opinion already made RISC-V the de facto standard open ISA.

RISC-V as its Berkeley designed RISC predecessors is a simple load-store instruction set designed for efficient compiler targeting and pipelining by maintaining a fixed instruction set encoding. The ISA is organized as four base instruction sets: RV32I, RV32E, RV64I and RV128I. RV32I is the base 32-bit integer instruction set with 32 general purpose registers. RV32E is a variation of RV32I with only 16 registers to make implementing low end embedded processors easier. RV64I and RV128I are the base 64-bit and 128-bit integer instruction sets which again have 32 64-bit and 128-bit general purpose registers respectively. There also exist many optional extensions for various use cases like the F and D extensions that add single and double precision floating point support, the M extension which adds multiplication and division capabilities, the A extension which adds atomic instructions for parallel programming and others [28], [8].

In this thesis we focus on the RV32IM and RV64IM ISAs since they are supported by our tools. RV64IM is a superset of RV32IM. Both ISAs' registers are shown in Table 1.1. The `x0` register has a constant value of zero to enable the synthesis of more complex instructions from simpler ones by the compiler. There also exist some special purpose registers that contain status and configuration bits. The integer data types supported by RISC-V are 8-bit bytes, 16-bit half-words, 32-bit words and 64-bit double words. RV32I can handle operations up to 32-bits whereas RV64I can handle operations up to 64-bits. Data types that do not completely fill the architectural registers are either zero or sign-extended depending on the signedness of the data. RISC-V supports only two data addressing modes namely immediate and displacement addressing. However through clever use of zeros the register indirect and limited absolute addressing modes can be implemented [28].

**Table 1.2: RV32I/64I user-visible registers [28], [8]**

| Register | Name | Use |
|---|---|---|
| `x0` | `zero` | Constant value 0 |
| `x1` | `ra` | Return address |
| `x2` | `sp` | Stack pointer |
| `x3` | `gp` | Global pointer |
| `x4` | `tp` | Thread pointer |
| `x5-x7` | `t0-t2` | Temporaries |
| `x8` | `s0/fp` | Saved register/Frame pointer |
| `x9` | `s1` | Saved register |
| `x10-x11` | `a0-a1` | Function arguments/return values |
| `x12-x17` | `a2-a7` | Function arguments |
| `x18-27` | `s2-s11` | Saved registers |
| `x28-x31` | `t3-t6` | Temporaries |

RISC-V instructions are encoded using a 32-bit fixed length encoding scheme making pipelining easier. The 6 possible instruction encodings can be seen in Figure 1.6. The `opcode` specifies the general type of instruction while the `funct` fields determine the specific operation. A 12-bit constant field is provided for displacement addressing, immediate constants or PC-relative addressing. The majority of RV32I and RV64I instructions can be seen in Table 1.2. Following the RISC paradigm instructions are simple and orthogonal to each other providing an efficient compiler target and relatively straight-forward hardware implementation.

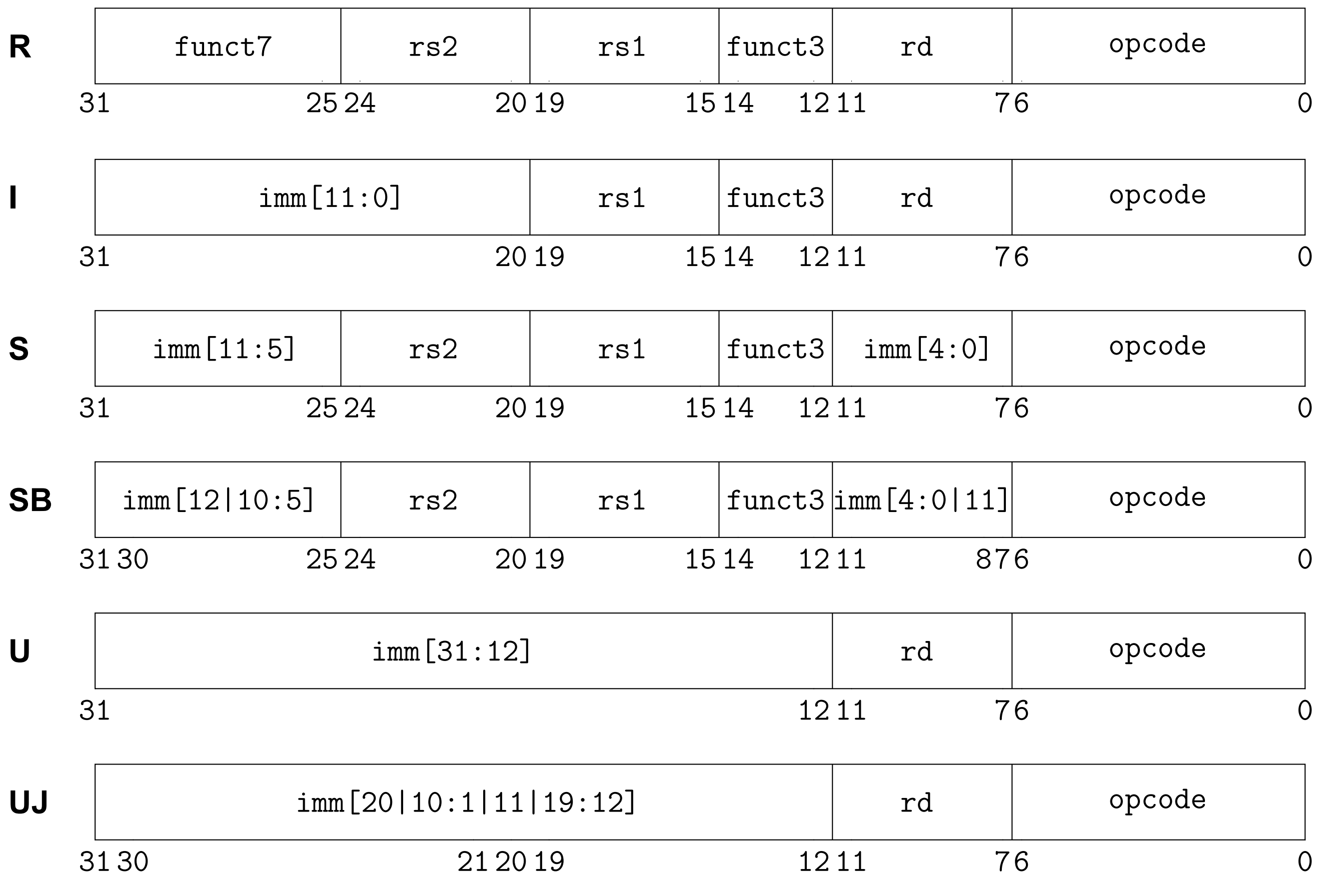


**Figure 1.6: Formats [28], [8]**

**Table 1.3: Instructions in RISC-V 32 and 64 bit version [28], [14]**

| Category | Functionality | Instruction | Format | Version |
|---|---|---|---|---|
| Data Transfer | Load byte | `lb rd, imm[11:0](rs1)` | I | RV32I |
| | Load byte unsigned | `lbu rd, imm[11:0](rs1)` | I | RV32I |
| | Load half | `lh rd, imm[11:0](rs1)` | I | RV32I |
| | Load half unsigned | `lhu rd, imm[11:0](rs1)` | I | RV32I |
| | Load word | `lw rd, imm[11:0](rs1)` | I | RV32I |
| | Load word unsigned | `lwu rd, imm[11:0](rs1)` | I | RV64I |
| | Load double word | `ld rd, imm[11:0](rs1)` | I | RV64I |
| | Store byte | `sb rs2, imm[11:0](rs1)` | S | RV32I |
| | Store half | `sh rs2, imm[11:0](rs1)` | S | RV32I |
| | Store word | `sw rs2, imm[11:0](rs1)` | S | RV32I |
| | Store double word | `sd rs2, imm[11:0](rs1)` | S | RV64I |
| Arithmetic/Logical | Add | `add rd, rs1, rs2` | R | RV32I |
| | Add 32 bits | `addw rd, rs1, rs2` | R | RV64I |
| | Sub | `sub rd, rs1, rs2` | R | RV32I |
| | Sub 32 bits | `subw rd, rs1, rs2` | R | RV64I |
| | Shift left logical | `sll rd, rs1, rs2` | R | RV32I |
| | Shift left logical 32 bits | `sllw rd, rs1, rs2` | R | RV64I |
| | Shift right logical | `srl rd, rs1, rs2` | R | RV32I |
| | Shift right logical 32 bits | `srlw rd, rs1, rs2` | R | RV64I |
| | Shift right arithmetic | `sra rd, rs1, rs2` | R | RV32I |
| | Shift right arithmetic 32 bits | `sraw rd, rs1, rs2` | R | RV64I |
| | And | `and rd, rs1, rs2` | R | RV32I |
| | Or | `or rd, rs1, rs2` | R | RV32I |
| | Xor | `xor rd, rs1, rs2` | R | RV32I |
| | Set if less than | `slt rd, rs1, rs2` | R | RV32I |
| | Set if less than unsigned | `sltu rd, rs1, rs2` | R | RV32I |
| | Add immediate | `addi rd, rs1, rs2` | I | RV32I |
| | Add immediate 32 bits | `addiw rd, rs1, rs2` | I | RV64I |
| | Shift left logical by immediate | `slli rd, rs1, rs2` | I | RV32I |
| | Shift left logical by immediate 32 bits | `slliw rd, rs1, rs2` | I | RV64I |
| | Shift right logical by immediate | `srli rd, rs1, rs2` | I | RV32I |
| | Shift right logical by immediate 32 bits | `srliw rd, rs1, rs2` | I | RV64I |
| | Shift right arithmetic by immediate | `srai rd, rs1, rs2` | I | RV32I |
| | Shift right arithmetic by immediate 32 bits | `sraiw rd, rs1, rs2` | I | RV64I |
| | And immediate | `andi rd, rs1, rs2` | I | RV32I |
| | Or immediate | `ori rd, rs1, rs2` | I | RV32I |
| | Xor immediate | `xori rd, rs1, rs2` | I | RV32I |
| | Set if less than immediate | `slti rd, rs1, rs2` | I | RV32I |
| | Set if less than immediate unsigned | `sliu rd, rs1, rs2` | I | RV32I |
| | Load upper immediate | `lui rd, imm[31:12]` | U | RV32I |
| | Add upper immediate to pc | `auipc rd, imm[31:12]` | U | RV32I |
| Control | Branch if equal | `beq rs1, rs2, imm[12:1]` | SB | RV32I |
| | Branch if not equal | `bne rs1, rs2, imm[12:1]` | SB | RV32I |
| | Branch if less than | `blt rs1, rs2, imm[12:1]` | SB | RV32I |
| | Branch if less than unsigned | `bltu rs1, rs2, imm[12:1]` | SB | RV32I |
| | Branch if greater or equal | `bge rs1, rs2, imm[12:1]` | SB | RV32I |
| | Branch if greater or equal unsigned | `bgeu rs1, rs2, imm[12:1]` | SB | RV32I |
| | Jump and link | `jal rd, imm[20:1]` | UJ | RV32I |
| | Jump and link register | `jalr rd, rs1, imm[11:0]` | I | RV32I |

## 1.6 Simulation Tools and Microprocessor Models

Simulation nowadays is used in all phases of hardware design and academic research, thus there exists a variety of simulation frameworks to choose from. Some widely used simulators are Simics, PTLsim, SimpleScalar, OVPsim, Spike, MARSSx86, Sniper and gem5 [18].

- *Simics* is a functional simulator that supports Alpha, ARM, MIPS, PowerPC, MSP430, SPARC and x86 architectures. It is often used to run production binaries meant for real hardware without modification [9].
- *PTLsim* is a cycle-accurate simulator designed for x86 microprocessors. More specifically, it targets a superscalar x86-64 OoO processor at varying levels of detail [44].
- *OVPsim* is a functional simulator that supports ARM, MIPS, PowerPC, x86 and RISC-V architectures including less known embedded ISAs [37].
- *Spike* is a functional simulator designed as a golden model for the RISC-V ISA. It comes packaged with RISC-V toolchain builds [8].
- *MARSSx86* is a cycle-accurate full-system simulator that supports multicore x86-64 systems. It also contains detailed models for coherent caches and on-chip interconnects [34].
- *Sniper* is a next-generation parallel x86-64 multicore simulator that uses the technique of interval simulation to achieve high throughput [27].
- *gem5* is a cycle-accurate simulator that supports the Alpha, ARM, PowerPC, x86, SPARC, MIPS and RISC-V ISAs. It is widely used in both academia and industry and provides a rich library of component models.

In our work we decided to use gem5 as it supports RISC-V and provides a cycle-accurate OoO microprocessor model. Furthermore, gem5 is the state-of-the-art simulator due to a wide and active community support, good documentation and mature code base while being stable, flexible and configurable [17].

When it comes to RTL microprocessor models, the RISC-V open ISA has enabled the development of open-source microprocessors that can be modified and used free of charge. This is beneficial to both academic research and commercial product development. Some prominent RISC-V microprocessor projects are Rocket, BOOM, CVA6 and RSD.

- *Rocket* is a 5-stage, in-order, scalar microprocessor that has OS support, a non-blocking data cache and front-end branch prediction. It implements the RV32G and RV64G ISAs where G symbolizes the base integer instruction set along with the M, A, F and D extensions [13].

- *BOOM* is an out-of-order, superscalar microprocessor with OS support that implements the RV64G ISA. It is considerably faster than Rocket even though it utilizes some components that where built for it. BOOM has an advanced branch predictor and uses aggressive out-of-order and superscalar techniques to achieve high performance [45], [13].
- *CVA6*, formerly known as Ariane, is a 6-stage, in-order, scalar microprocessor with OS support and front-end branch prediction. It implements the RV64IMAC ISA [22].
- *RSD* is an open-source out-of-order, superscalar processor optimized for FPGA use. It implements aggressive out-of-order and speculative execution features to elevate performance. Even with this advanced feature set it manages to keep area down especially when implemented on FPGAs. RSD implements the RV32IM ISA and doesn't have OS support [33].

After comparing the available microprocessors we decided on RSD. Our choice was mainly based on the fact that RSD had a more compact code base than other choices while supporting OoO execution. Being a size optimized design simulation is faster on RSD than most other choices thus making experimentation easier. Furthermore, Professor Ryota Shioya, a main contributor to the RSD project, was always available for support during our research thus saving a lot of troubleshooting time.

# 2. TOOLS AND MODELS FULL SUMMARY

## 2.1 RSD

Out-of-order execution, otherwise known as dynamic scheduling, increases the amount of instruction level parallelism by reducing avoidable stall cycles. RSD implements OoO execution in combination with speculative scheduling to increase performance. Studies have shown that speculative scheduling can increase the Instructions per Cycle (IPC) of SPECint2006 by 26.8% [33]. The RSD processor also supports OoO load and store execution and disambiguation, a memory dependence predictor and a non-blocking data cache. In combination with these features, RSD is highly optimized for FPGA synthesis by using native FPGA resources whenever possible thus more efficiently utilizing the FPGA fabric. In this section we will take a deep dive into the RSD microarchitecture and finally present the exact RSD configuration that was used for the purposes of this study.

We can divide RSD into three logical blocks. The Front-End Block, the Scheduling Block and the Execution Block.

- The *Front-End Block* fetches and decodes instructions from the level 1 instruction cache in program order. The gshare branch predictor is used to predict the direction of branches by combining the global branch history and the location of the executed branch thus increasing prediction accuracy [26].
- The *Scheduling Block* is responsible for maximizing instruction level parallelism (ILP) for instructions sent from the front-end block. To do this it issues instructions to the execution block out of the program order. The main components of the scheduling block are the rename unit, the dispatch unit, the issue queue (IQ) and the reorder buffer (ROB)
  - The *Rename Unit* removes false dependencies between instructions by renaming the architectural registers corresponding to destination operands to registers in the physical register file. This is achieved through the use of a register map table (RMT).
  - The *Dispatch Unit* reserves an entry for renamed instructions in components like the ROB, the load queue (LDQ) and the store queue (STQ), of course depending on the type of instruction. If any of the aforementioned units is full and thus an entry can't be allocated for an instruction the dispatch unit stalls until an entry is freed up.
  - The *Issue Queue* issues instructions to the execution block when the source operands of the instruction are available. It is comprised of the wakeup logic, the select logic and the instruction payload RAM. The wakeup logic determines the readiness of each in-flight instruction enabling the select logic to issue ready instructions to the execution block. The instruction payload RAM stores data required for executing the instruction like the instruction's type. Going into a bit

more detail, the wakeup logic uses a matrix based approach where asserted matrix bits signify that the instruction corresponding to the matrix row depends on the instruction corresponding to the matrix column. The select signal clears an instructions column bits when it is selected and issued. The select logic considers an instruction ready when all bits of the corresponding row are cleared.

  - The *Reorder Buffer* stores the state of dispatched instruction and commits an instruction when it finishes executing and becomes the oldest instruction in the processor. This ensures that the programmer perceives the execution of instructions as in-order maintaining program correctness. If an instruction is executed erroneously due to mis-speculation then the instruction and its successors are flushed or replayed.

- The *Execution Block* executes the instructions it receives from the scheduling block. Its main components are the physical register file (PRF), the load-store unit (LSU) and the functional units themselves.

  - The *Physical Register File* stores operand physical register data. Instructions once at the PRF first receive their source operand data from it or directly from the execution units through a bypass network. The appropriate execution unit then executes the instruction and writes the result to the PRF and the bypass network.
  - The *Load Store Unit* handles memory instructions and acts as a bridge to the DRAM system by sending a cache fill request when a memory instruction misses in the level 1 cache. RSD's LSU allows loads and stores to be executed out-of-order speculatively significantly reducing the memory bottle-neck. The correctness of memory instructions is guaranteed by performing dynamic memory disambiguation through the use of load and store queues. These queues are content addressable memory (CAM) structures that are filled with loads and stores respectively. Dynamic memory disambiguation is performed by having loads search for older stores to the same address in the store queue and using the youngest saved data. Stores on the other hand search for younger completed loads from the same address in the load queue and force instruction replays for them and their dependents. In an effort to minimize replays due to memory ordering violations, RSD uses a memory dependence predictor (MDP) which forces dependence predicted loads to wait in the IQ for all older stores to complete.

As mentioned above, RSD supports speculative scheduling. This means that instructions that depend on other instructions of variable execution time, such as loads, are issued on the assumption that the latter instruction will complete with the minimum possible latency. In the case of loads this can significantly decrease the load-to-use latency, contributing to better performance. The instruction replay mechanism is used when an erroneous scheduling decision is made meaning that the source operand data has not yet been produced by the variable latency instruction. RSD uses an instruction queue based replay design where speculatively scheduled instructions are kept in the IQ and the IQ replay logic interacts with the wakeup logic to replay instructions when necessary.

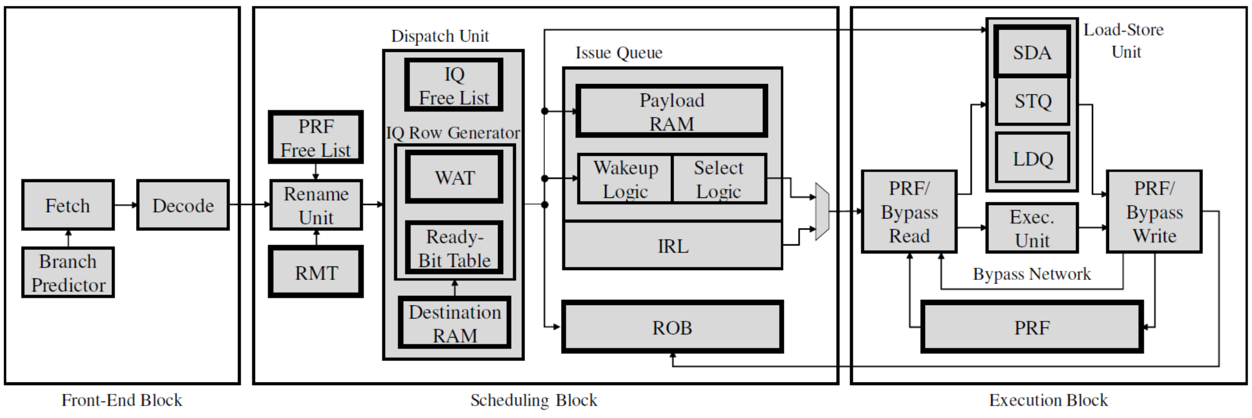


**Figure 2.1: RSD Block Diagram [33]**

**Table 2.1: Main Microarchitectural Parameters of RSD**

| Parameter | Value |
|---|---|
| Pipeline | OoO |
| Fetch/Decode/Rename Width | 2 |
| Branch Predictor | gshare (2048 History Table) |
| Branch Target Buffer Entries | 1024 |
| Return Address Stack Entries | 4 |
| Issue Width | 5 |
| Writeback Width | 5 |
| Commit Width | 2 |
| Physical Register File | 64 Registers |
| Instruction Queue Entries | 16 |
| Reorder Buffer | 64 |
| L1 Data/Instruction Cache | 4kB/4kB (2-way) |
| Cache Line Size | 8 bytes |
| Replacement Policy | Tree-PLRU |
| L1 Hit Latency | 1 clock cycle |
| L1 Miss Latency | 100 clock cycles |
| MSHR Entries | 2 |
| Load/Store Queue Entries | 16 |

Figure 2.1 shows the block diagram of RSD. The flow of instructions through the three main RSD blocks can clearly be seen. The advanced features that RSD supports in combination with the good documentation, compact code base and excellent support from Professor

Shioya make RSD the best choice for our study. We hope that our current observations and feedback can help the RSD team improve their design even more as they have already done based on previous suggestions.

Table 1.3 presents the main microarchitectural parameters of RSD. We can see that RSD is a 2-way fetch superscalar core with an issue width of five to account for the five available execution units i.e two integer units which handle simple integer and logical instructions as well as branches, a complex integer unit which handles multiplication and division, a load unit and a store unit. The physical register file contains 64 registers and the ROB can hold 64 entries. The instruction queue can hold 16 instructions at a time and the load and store queues can hold 16 entries each. The branch predictor used is gshare with a PHT of 2048 entries and a BTB of 1024 addresses. The L1 instruction and data caches are 2-way set associative with a size of 4kB and 2 MSHRs[5].

## 2.2 Verilator

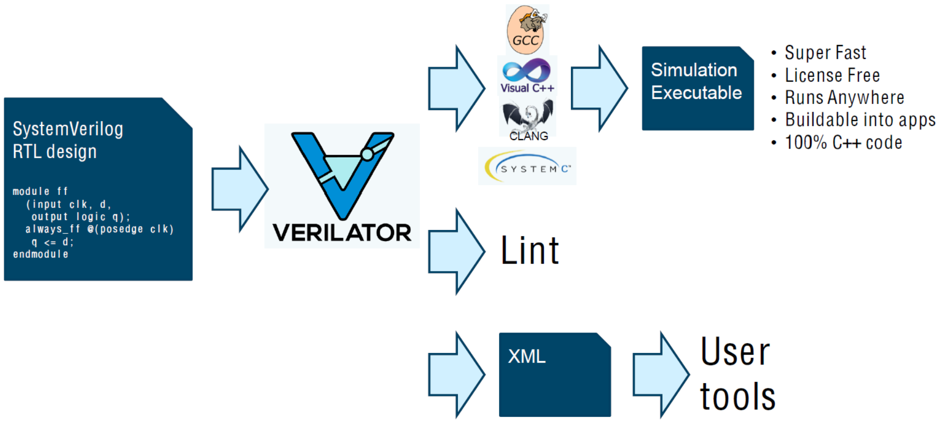


**Figure 2.2: Verilator Flow [41]**

There exist many HDL simulators that one can use when performing behavioral RTL simulation. The most widely used commercial simulators are Mentor Graphics ModelSim, Synopsys VCS and the Cadence Incisive Enterprise Simulator. These all support both VHDL and (System)Verilog and have been adopted by many IC design and testing companies. Some well known open-source simulators are Icarus Verilog, Verilator and GHDL. As their names imply Icarus Verilog and Verilator support Verilog whereas GHDL supports VHDL. Open-source simulators are generally used for smaller projects but with the rise of the open-hardware movement they are being utilized in larger and larger endeavours [42], [43], [41].

In order to perform behavioral RTL simulation of the RSD processor Verilator is used. Verilator is an open-source, high throughput, widely used (System)Verilog simulator. It accepts synthesizable Verilog or SystemVerilog code and after performing lint code-quality

[5]MSHRs keep track of outstanding cache misses and pending load/store accesses that refer the missing cache line

checks it compiles it into multithreaded C++ code as can be seen in Figure 2.2. According to [41] Verilator outperforms many commercial simulators while supporting both single and multi-threaded output models. Verilator is widely used in both academia and industry and has out-of-the-box support for ARM and RISC-V vendor IP.

Once Verilator has created an optimized C++ model from the supplied RTL code the user writes a C++ wrapper that instantiates said model of the top level module. The C++ files are then compiled using a conventional C++ compiler like GCC or Clang. The produced executable performs the simulation. Due to the high level of code optimization the Verilator model is about 100x faster than interpreted Verilog and by making use of multithreading this speedup can be extended to 200x up to 1000x. The fact that Verilator is free and open-source while being able to compete with and often surpass the performance of commercial simulators makes it the best choice value-wise for our academic endeavour.

## 2.3 Konata

In this thesis except from traditional debugging techniques such as microarchitectural counter comparison and step-by-step execution we utilized Konata, a free and open-source pipeline visualizer developed by Professor Ryota Shioya. Konata enables the visualization of O3PipeView compatible trace files in an easy to navigate graphical user interface. Features such as side by side comparison of trace files and aggregated statistics information are very useful when trying to match the pipeline behavior of two different models. The ability to graphically represent the execution flow of different workloads makes finding flushes or cache misses much easier and is a quick way to compare performance [38]. Figure 2.3 shows a sample trace file being visualized in Konata.

## 2.4 gem5

gem5 is an open-source simulation tool widely used in computer architecture research. It offers stability and flexibility that enable computer engineers to execute applications in multiple architectures and evaluate hardware at the cycle level. gem5 is known for its extensive and supportive community and its frequent updates that consistently add new features and improve already existing ones. The gem5 simulator supports a variety of ISAs including RISC-V, it has two types of execution modes and contains multiple CPU models and advanced protocols for caches and interconnection networks [31], [17].

The flexibility that gem5 provides is in part owed to the modular architecture of component definitions. Developers represent each component with two classes, one in Python and the other in C++. The Python class contains all the component parameters which can easily be modified by the end user whereas the C++ class determines the component's functionality. As a result of this, all gem5 components, e.g CPU and memory models have familiar structure and initialization flow.

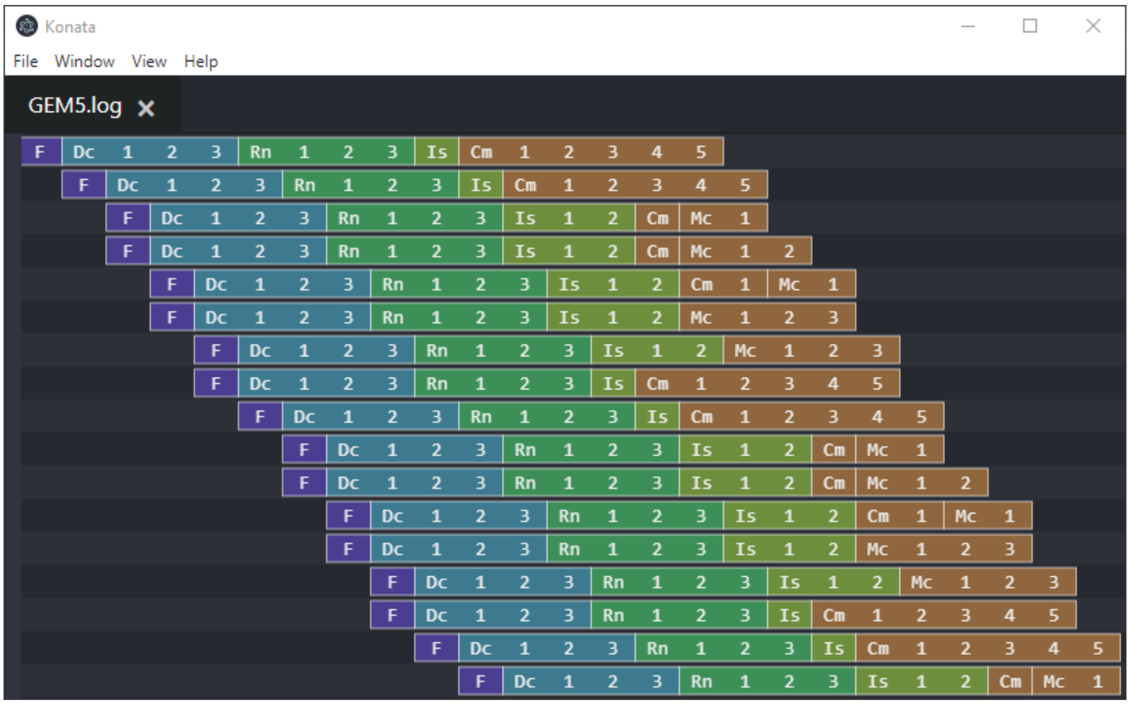

Konata
File Window View Help
GEM5.log
F Dc 1 2 3 Rn 1 2 3 Is Cm 1 2 3 4 5
F Dc 1 2 3 Rn 1 2 3 Is Cm 1 2 3 4 5
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm Mc 1
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm Mc 1 2
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm 1 Mc 1
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Mc 1 2 3
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Mc 1 2 3
F Dc 1 2 3 Rn 1 2 3 Is Cm 1 2 3 4 5
F Dc 1 2 3 Rn 1 2 3 Is Cm 1 2 3 4 5
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm Mc 1
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm Mc 1 2
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm 1 Mc 1
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Mc 1 2 3
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Mc 1 2 3
F Dc 1 2 3 Rn 1 2 3 Is Cm 1 2 3 4 5
F Dc 1 2 3 Rn 1 2 3 Is Cm 1 2 3 4 5
F Dc 1 2 3 Rn 1 2 3 Is 1 2 Cm Mc 1

**Figure 2.3: Konata Sample Trace File Visualization**

While selecting the execution mode, CPU model and cache protocol, the researcher must consider the trade-off between accuracy and speed, as shown in Figure 2.4. gem5 has two system modes, *System-call Emulation* (SE) mode and *Full-System* (FS) mode. In System-call Emulation mode, gem5 doesn't load an operating system and as a result the system calls are emulated through the host OS. On the other hand, in Full System mode, gem5 creates a bare-metal environment configured to run an OS which includes support for interrupts, exceptions, privilege levels and I/O devices. In this mode gem5 executes (simulates) both user and kernel-level instructions without deferring to the host machine support.

The choice of CPU model also plays an important role in simulation accuracy and speed. According to [31], gem5 supports four main CPU models:

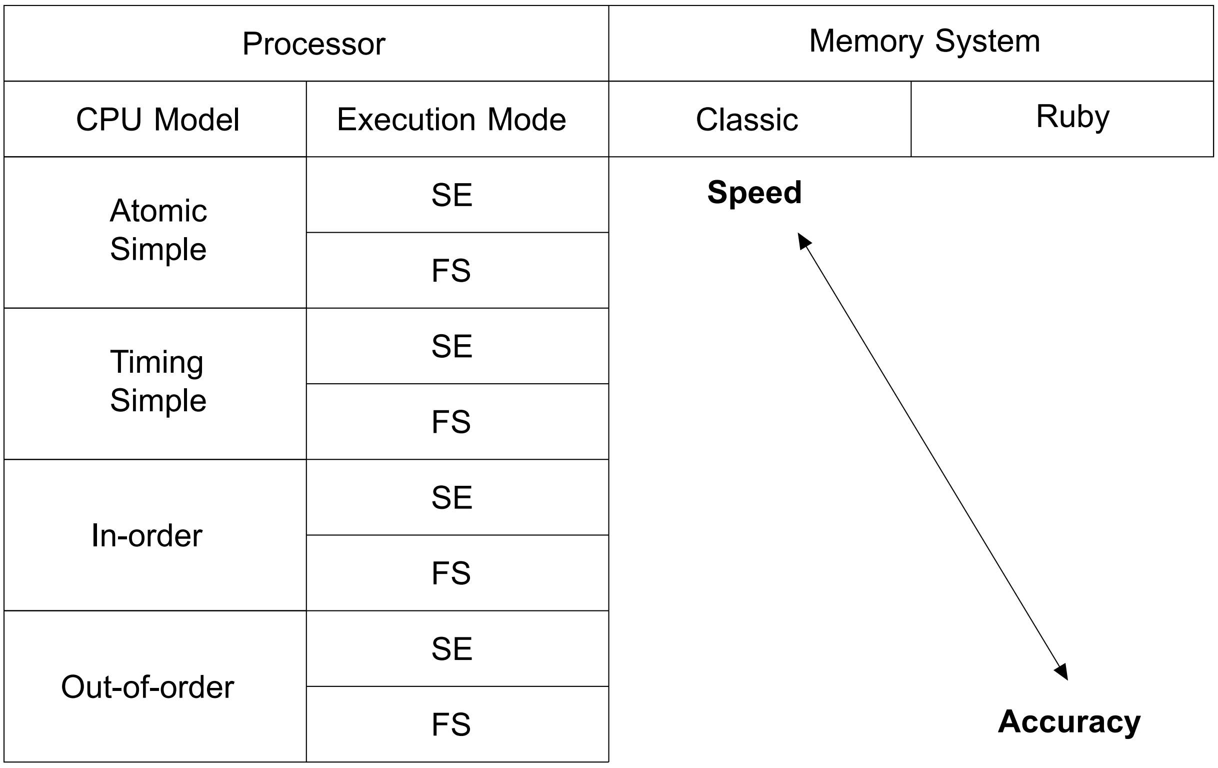
i. *Simple* CPU (Atomic and Timing) are non-pipelined models that can be used for memory system studies and studies that do not require high execution fidelity. These models fetch, decode, execute and commit a single instruction every cycle. Simulation throughput is high due to the simplicity of these models.

ii. The *In-order* CPU model simulates an in-order pipelined machine with a configurable number of pipeline stages and other parameters such as issue width. Instruction execution takes place exclusively in the execution stage after all dependencies have been resolved ("execute-in-execute" model) [17].

iii. The *O3* model simulates an out-of-order pipelined CPU. It includes parameterizable functional units such as load and store queues and a reorder buffer making superscalar execution feasible. It is also an “execute-in-execute” model.

iv. The *Kernel Virtual Machine* (KVM) CPU is based on the KVM API in Linux and allows gem5 to utilize the host’s processor in order to execute programs. It requires the same ISA to be used in gem5 and the host machine.

Furthermore, gem5 supports a variety of cache configurations. Classic cache configurations include two level, three level and non-coherent cache hierarchies with write-through and write-back policies and Ruby Cache Models include MOESI Two Level and MESI Three Level coherent caches. The Ruby memory system is easily modifiable and can simulate advanced cache hierarchies with many replacement policies.



| Processor | | Memory System | |
|---|---|---|---|
| CPU Model | Execution Mode | Classic | Ruby |
| Atomic Simple | SE | | |
| | FS | | |
| Timing Simple | SE | | |
| | FS | | |
| In-order | SE | | |
| | FS | | |
| Out-of-order | SE | | |
| | FS | | |

**Figure 2.4: Speed vs Accuracy in gem5 [17]**

In our work we chose the SE execution mode and O3 CPU model in order to approach RSD’s features and characteristics. The choice of the SE execution mode is most logical since OS support is missing from the current RSD implementation. The O3 pipeline stages in gem5 are five: fetch, decode, rename, issue/execute/writeback (IEW) and commit [2].

i. *Fetch*. Fetches instructions and performs branch prediction.

ii. *Decode*. Decodes instructions each cycle and handles unconditional control flow instructions.

iii. *Rename*. Renames instructions' architectural registers to physical registers and stalls if there are not enough registers to rename to.

iv. *IEW*. Removes an instruction from the issue queue, executing it and writes back the result.

v. *Commit*. Commits instructions each cycle and handles any faults/exceptions that the instructions may have caused.

The O3 pipeline has configurable widths and inter-stage delays. Further system customization can be achieved by choosing a custom branch predictor and changing the memory system that is used.

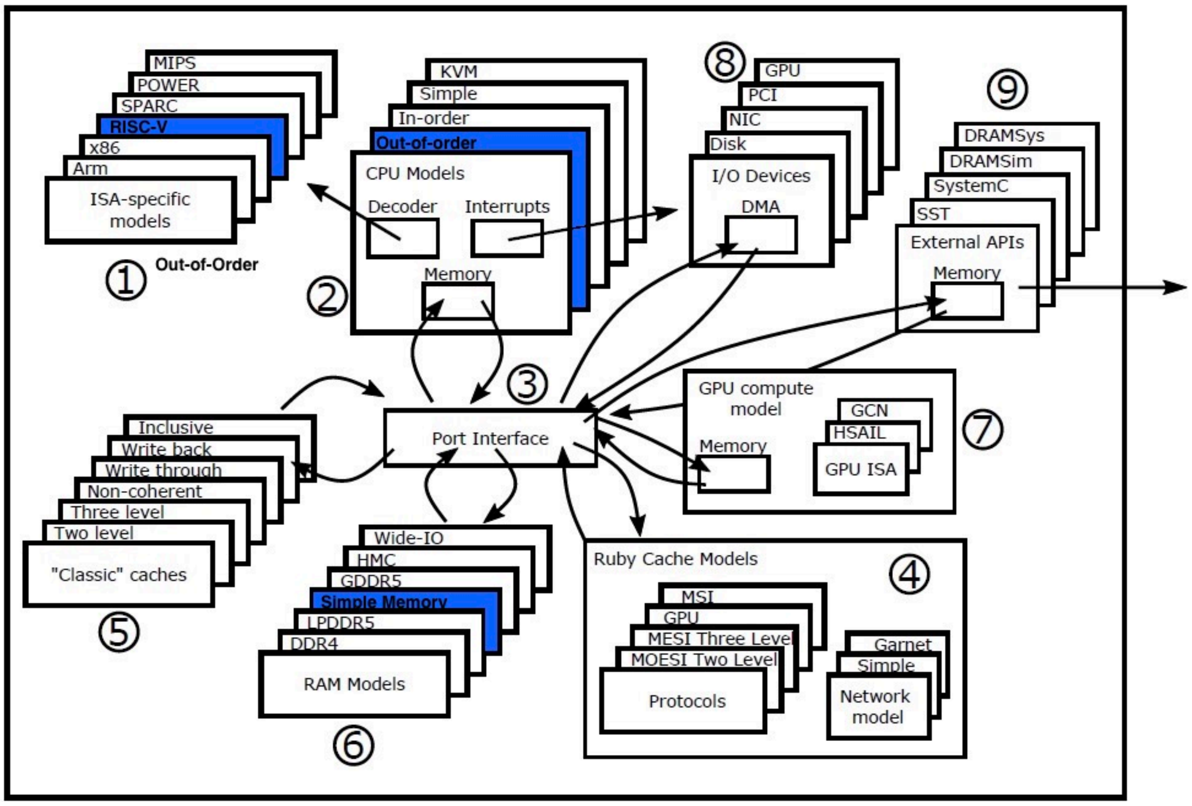


**Figure 2.5: gem5 Block Diagram [31]**

In summary, gem5 offers a variety of CPU models, two execution modes and different memory system models and cache hierarchies to choose from. These features make it a very useful and flexible tool for computer architecture research. The level of customization that gem5 provides us with enabled us to make fine adjustments to each component in our effort to match gem5's performance to that of the RSD. In this study we utilize the out-of-order CPU model running in the SE mode which provides very good accuracy as seen in Figure 2.4 while maintaining a solid speed advantage over RTL simulation.

# 3. EXPERIMENTAL METHODOLOGY

In this chapter we present the steps that we followed in our effort to match gem5's to RSD's performance. We aim to benefit from the high simulation throughput of gem5 while maintaining relatively high accuracy when compared to RTL simulation. To do this our approach is to fine-tune the gem5 O3 CPU model and memory system in order to closely match RSD. We will commence with tool installation and continue with microarchitectural parameter discovery, microbenchmarking and finally present all the modifications made to gem5.

In our work we used version 20.1.0.4 of gem5 and the fix-branch-predictor (last update 24 March 2020) branch of RSD. The fix-branch-predictor branch was used instead of the master branch under the direction of Professor Shioya as it fixes some bugs in the gshare implementation. To compile test programs and (micro-)benchmarks we compiled the riscv-gnu-toolchain repository for bare-metal RV32IM and RV64IM targets. The toolchain includes a cross-compiler and corresponding debugging software. To run the behavioral RTL simulation of RSD version 4.106 of Verilator was used. By installing the corresponding versions of the specified tools our work can easily be replicated after following the steps outlined in this chapter.

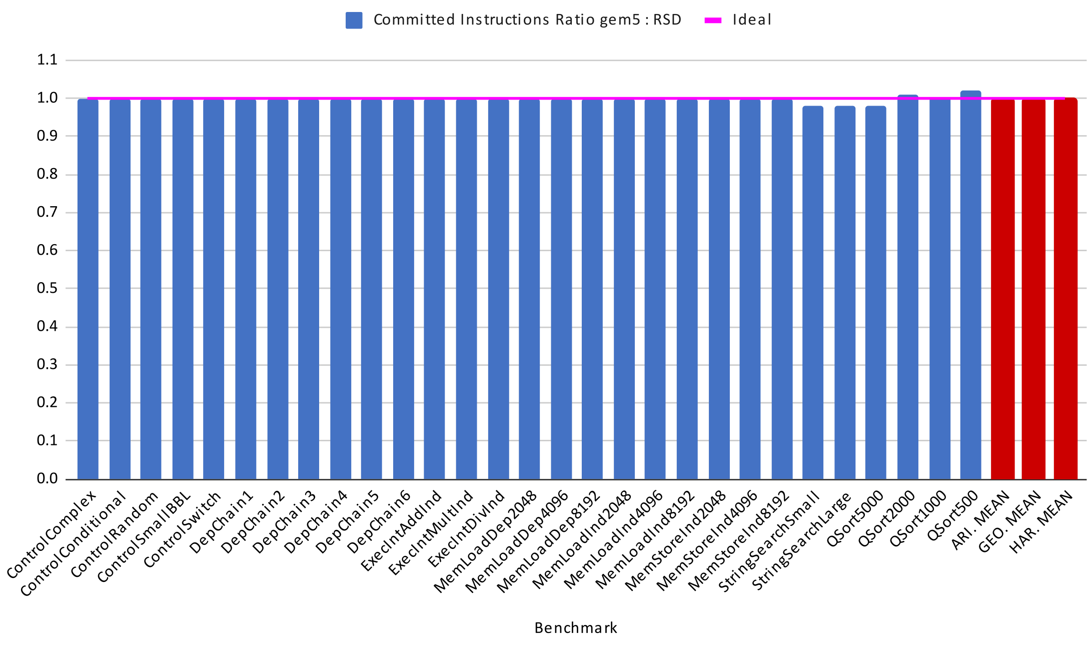


**Figure 3.1: Committed Instructions Ratio** $gem5 : RSD$

As mentioned above we compiled the RISC-V cross-compiler to support both RV32IM and RV64IM ISAs. The reason why we need both 32 and 64-bit support is because RSD implements the 32-bit RISC-V ISA whereas gem5 only supports the 64-bit version. This does not pose a major issue for the purposes of our study (i.e performance modeling) as

RV64IM is a proper superset of RV32IM i.e it contains all instructions that are present in RV32IM. The main difference between the two ISA variations is the width of the registers used. RV64IM has 64-bit registers and thus natively handles 64-bit data (e.g the `add` instruction calculates the sum of two 64-bit integers). When it comes to handling 32-bit data RV64IM utilizes additional instructions whose assembly mnemonic includes a `w` suffix (e.g `addw`). Depending on the signedness of the data the compiler may introduce additional instructions for sign extension. In order to avoid introducing unwanted sources of error when measuring the performance of our programs we decided to use the native data types for each ISA. The effectiveness of this approach is portrayed in Figure 3.1 which shows that committed instructions in both simulations are virtually identical. The absolute error is only 0.32%. The functionality and purpose of the used benchmarks will be described in detail later in this chapter.

The first major step in fine tuning our microarchitectural model was to determine the essential parameters of RSD's microarchitecture. Since RSD is an open-source project this process was significantly simplified as most parameters where available in SystemVerilog code. After carefully parsing the source code we where able to uncover most of them giving us a good picture of RSD's design. Table 2.1 includes the most important parameters of RSD's microarchitecture. In order to match these to gem5 equivalent parameters we also parsed the relevant gem5 source code and configuration files in conjunction with careful reading of gem5's documentation. A final check to confirm our choices was done by Professor Shioya who has experience using both gem5 and RSD.

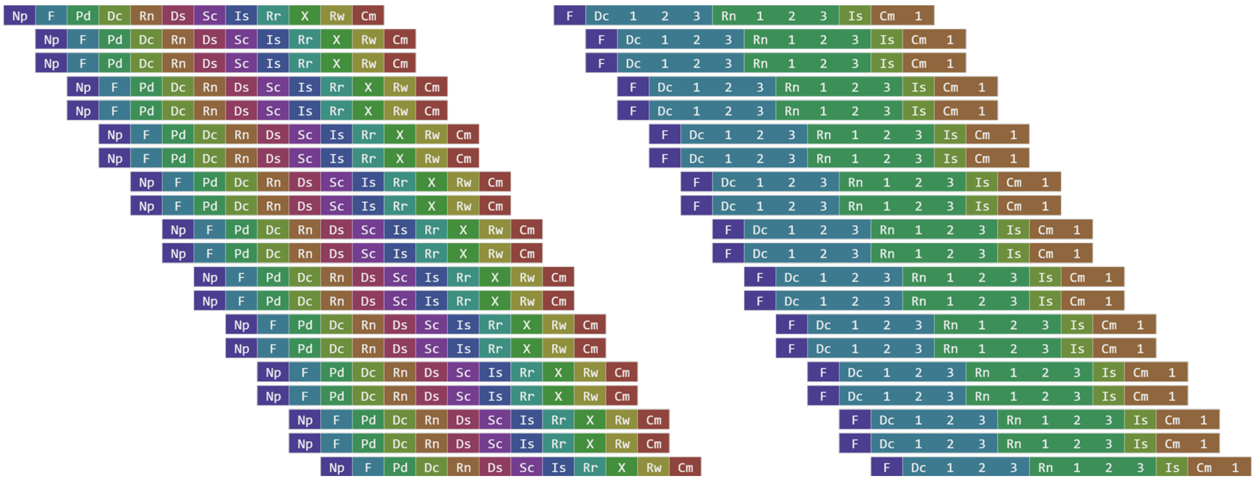

Np F Pd Dc Rn Ds Sc Is Rr X Rw Cm
F Dc 1 2 3 Rn 1 2 3 Is Cm 1

**Figure 3.2: RSD (*left*) - gem5 (*right*) Pipeline Matching**

As seen in Table 2.1 RSD uses the gshare branch predictor. Gshare uses a register to hold the branch global history i.e a string of bits that indicate whether previous branches were taken or not taken. This history is combined with the PC through the use of a hash function generating an index to access a table of 2-bit saturating counters. The motivation for using gshare and in fact any correlating branch predictor is to utilize a different finite state machine (FSM) for every combination of global history and branch address. Since this is not possible to implement ideally in practice, some degree of aliasing will appear.

The use of an exclusive-or operation between the lower bits of the PC and the branch history have been shown to minimize the probability of aliasing [25]. Since gshare is not included in the library of gem5 components we had to search for an open implementation of gshare outside the bounds of gem5's library. After comparing available models we decided upon [36]. After modifying the provided code to suit our specific needs we added it into our gem5 local repository. The new component provided us with configurable prediction history table entries and saturating counter widths which we adjusted according to Table 2.1.

Another major aspect of configuring the O3 CPU model in gem5 was to correctly set up the pipeline length. At first since gem5 by default uses an idealized pipeline configuration with 5 stages including a combined Issue-Execute-Writeback stage the difference between the two models was vast thus resulting in significantly different performance from RSD. In order to fix this issue we adjusted the decode-to-rename and rename-to-IEW delays so that the total pipeline length matched that of RSD. Additionally the specific number of pipeline stages that different execution units need to complete their calculations where matched to those of RSD through the modification of the functional unit configuration files in gem5. The pool of available functional units was also modified to only include the 2 integer, 1 complex integer and 2 memory load and store units of RSD. After making said changes and running a test integer stress mark the matching pipeline behavior can be observed in Figure 3.2. In order to get a detailed view of both pipelines the Konata pipeline visualizer was used throughout this matching effort. After making modifications to various delay parameters the above configuration provided the best correlation between gem5 and RSD.

In order to test the effect of the changes we made to gem5 gauging how close we could match its O3 CPU model to RSD we utilized a variety of general purpose benchmarks and targeted microbenchmarks. After performing a thorough literature review and also designing a few custom programs we decided upon the use of the microbenchmarks described in [11] as they feature a streamlined design and have already been used for simulator validation. In conjunction with those we also ran a few general purpose benchmarks from the MiBench suite. The lack of more complex and extensive benchmarks is mainly attributed to the small memory size of RSD and its lack of support for an OS. Even without running a larger and more standardized test suite we believe that the programs we used adequately stress the two models and are representative of real world performance.

The chosen microbenchmark suite includes four types of microbenchmarks: Control, Dependency, Execution and Memory [11], described briefly in the Table 3.1.

i. *Control* microbenchmarks stress the execution flow of the microprocessor mainly stressing the branch predictor. More specifically

   - *Control Conditional* includes an if-then-else statement in a loop and alternates between taking and not taking the resulting conditional branch.
   - *Control Switch* produces many indirect jumps via the use of a switch statement with 10 case statements in a loop. Each case statement is taken $n/10$ times on

consecutive loop iterations before moving to the next case statement ($n$ represents the number of loop repetitions).

- *Control Complex* combines if-else and switch constructs in order to make branch prediction more difficult.
- *Control Random* includes an if-then-else statement in a loop whose resulting conditional branch depends on the value of a Linear Feedback Shift Register (LFSR). Since the branch direction is decided randomly this microbenchmark aids in determining the branch miss penalty.
- *Control Small BBL* assesses the number of simultaneous in-flight branches as it executes a single loop with only one instruction inside that increments the loop counter.

ii. *Dependency* microbenchmarks stress dependency forwarding between instructions. They evaluate chains of instructions with lengths ranging from 1 to 6. Each instruction is dependent on the data produced by its predecessor thus evaluating the data forwarding time. The processor component being stressed here is the physical register file and its forwarding paths.

iii. *Execution* microbenchmarks test the functional units of the CPU. In our work we utilize the integer add, integer multiply and integer divide stressmarks. Each program includes 32 independent instructions inside a loop in order to minimize the number of memory operations, control hazards and data dependencies thus allowing for almost ideal throughput.

iv. *Memory* microbenchmarks stress the cache and memory hierarchy. More specifically

- *Load Dependent* implements a loop that traverses a linked list having to wait for each load to complete before starting the next. Three different linked list sizes are used. The two smaller linked list size versions stress the L1 data cache and the larger size version stresses the main memory.
- *Load Independent* executes 32 parallel independent loads from an array and sums up the resulting values inside a loop. Again three different array sizes where used where the two smaller ones stress the L1 data cache and the larger stresses the main memory.
- *Store Independent* performs 32 parallel independent stores in a loop iterating over all positions of an array. The same three array sizes as before where used to stress both the L1 data cache and main memory.

As mentioned above in addition to the microbenchmark collection we also executed some general purpose benchmarks from the MiBench suite. These include the StringSearchLarge, StringSearchSmall and QSort benchmarks. For QSort we also varied the input array length in order to test for the sensitivity of the microarchitectural model to input size changes. StringSearchLarge and StringSearchSmall search for specified words in sentences using a case insensitive comparison algorithm. QSort sorts an array of strings into ascending order using the quick sort sorting algorithm.

## Table 3.1: gem5 Modified Parameters

| RSD Parameter | gem5 Equivalent Parameter | Value | Explanation |
|---|---|---|---|
| Fetch Width | `O3CPU.py/fetchWidth` | 2 | RSD is a 2-way fetch superscalar processor with 5 functional units |
| Decode Width | `O3CPU.py/decodeWidth` | 2 | |
| Rename Width | `O3CPU.py/renameWidth` | 2 | |
| Dispatch Width | `O3CPU.py/dispatchWidth` | 2 | |
| Issue Width | `O3CPU.py/issueWidth` | 5 | |
| Write-Back Width | `O3CPU.py/wbWidth` | 5 | |
| Commit Width | `O3CPU.py/commitWidth` | 2 | |
| N/A | `O3CPU.py/squashWidth` | 2 | |
| N/A | `O3CPU.py/decodeToRenameDelay` | 4 | Experimentally modified to match RSD's pipeline length |
| N/A | `O3CPU.py/renameToIEWDelay` | 4 | |
| Reorder Buffer Entries | `O3CPU.py/numROBEntries` | 64 | Match RSD |
| Issue Queue Entries | `O3CPU.py/numIQEbtries` | 16 | Match RSD |
| Physical Integer Register Number | `O3CPU.py/numPhyIntRegs` | 64 | Match RSD |
| Fetch Buffer Size | `O3CPU.py/fetchBufferSize` | 8 | Equal to the cacheline size |
| Fetch Queue Size | `O3CPU.py/fetchQueueSize` | 2 | RSD does not have a fetch queue |
| Branch Predictor | `O3CPU.py/branchPred` | gshare | RSD uses gshare with a PHT of 2048 entries and a BTB of 1024 entries and tag size of 4 bits. The RAS holds 4 entries |
| Local Predictor Size | `BranchPredictor.py/localPredictorSize` | 2048 | |
| Branch Target Buffer Entries | `BranchPredictor.py/BTBEntries` | 1024 | |
| Branch Target Buffer Tag Size | `BranchPredictor.py/BTBTagSize` | 4 | |
| Return Address Stack Entries | `BranchPredictor.py/RASSize` | 4 | |
| Load Queue Entries | `O3CPU.py/LQEntries` | 16 | Match RSD |
| Store Queue Entries | `O3CPU.py/SQEnries` | 16 | |
| Integer Functional Units | `FuncUnitConfig.py/IntALU.count` | 2 | Match RSD's FU configuration |
| Complex Integer Functional Units | `FuncUnitConfig.py/IntMultDiv.count` | 1 | |
| Divide Latency | `FuncUnitConfig.py/IntMultDiv.intDiv.opLat` | 32 | |
| Load and Store Functional Units | `FuncUnitConfig.py/RdWrPort.count` | 2 | |
| Load Unit Latency | `FuncUnitConfig.py/RdWrPort.MemRead.opLat` | 3 | |
| Store Unit Latency | `FuncUnitConfig.py/RdWrPort.MemWrite.opLat` | 3 | |
| Cache Load Ports | `O3CPU.py/cacheLoadPorts` | 1 | Match RSD's cache configuration |
| Cache Store Ports | `O3CPU.py/cacheStorePorts` | 1 | |
| Instruction Cache Size | `--l1i_size`[6] | 4kB | |
| Instruction Cache Associativity | `Cashes.py/assoc` | 2 | |
| Data Cache Size | `--l1d_size`[6] | 4kB | |
| Data Cache Associativity | `Cashes.py/assoc` | 2 | |
| Cache Line Size | `--cacheline_size`[6] | 8B | |
| Cache MSHRs | `Cashes.py/mshrs` | 2 | |
| Cache Targets per MSHR | `Caches.py/tgts_per_mshr` | 1 | |
| Instruction Cache Tag Latency | `Cashes.py/tag_latency` | 1 | |
| Instruction Cache Data Latency | `Cashes.py/data_latency` | 1 | |
| Instruction Cache Response Latency | `Cashes.py/response_latency` | 1 | |
| Replacement Policy | `Cache.py/replacement_policy` | TreePLRURP | |
| Memory Latency | `SimpleMemory.py/latency` | 100ns | Match RSD's fixed memory access time (simulation only) |

[6]These are given as command line parameters to the `se.py` script after the `--caches` directive

**Table 3.2: Micro-architectural Units Stressed by Each Benchmark [11]**

| | | Control | | | | | Dependency | | | | | | Execution | | | Memory | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Conditional | Switch | Complex | Random | Small BBL | Chain 1 | Chain 2 | Chain 3 | Chain 4 | Chain 5 | Chain 6 | Integer Add | Integer Mult | Integer Div | Load Ind 2048 | Load Ind 4096 | Load Ind 8192 | Load Dep 2048 | Load Dep 4096 | Load Dep 8192 | Store Ind 2048 | Store Ind 4096 | Store Ind 8192 |
| **Characteristics** | Branch Predictor | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | | | | | | | | | | | | |
| | Branch Miss Penalty | | | | ✓ | | | | | | | | | | | | | | | | | | | |
| | In-flight Branches | | | | | ✓ | | | | | | | | | | | | | | | | | | |
| | Register File | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | | | | | | |
| | Functional Units | | | | | | | | | | | | ✓ | ✓ | ✓ | | | | | | | | | |
| | L1 Cache | | | | | | | | | | | | | | | ✓ | ✓ | | ✓ | ✓ | | ✓ | ✓ | |
| | Memory | | | | | | | | | | | | | | | | | ✓ | | | ✓ | | | ✓ |

# 4. RESULTS AND ANALYSIS

In this chapter we are going to present the results of our effort to match gem5's to RSD's performance. Our main error metric will be the cycles ratio between the two models which ideally should be 1 (i.e a perfect match). We will also calculate the arithmetic, geometric and harmonic mean of the cycles ratio of all our programs and the average absolute error i.e

$$\frac{1}{n}\sum_{i=1}^{n}\left|1-\left(\frac{gem5\ cycles}{RSD\ cycles}\right)_i\right| \tag{4.1}$$

where $n$ is the total number of benchmarks.

As we can see in Figure 4.2 the cycles ratio of 20 out of 29 programs is close to 1. The arithmetic mean is 1.01, the geometric mean is 0.95 and the harmonic mean is 0.88. Using Equation 4.1 we calculate that the average absolute error across all benchmarks is 21.77%. The large discrepancy observed when running ExecIntDivInd is due to the fact that gem5 uses a fixed latency divider whereas RSD uses a variable latency one. This is not representative of real-world performance as divide instructions are not very common. The MemStoreInd8192 microbenchmark also represents an edge case where continuous store instructions are executed all of which result in a cache miss thus also not being indicative of real-world performance. After removing the ExecIntDivInd and the MemStoreInd8192 microbenchmarks the absolute error drops down to 15.35% which is inline with other validation studies [11] (geometric mean of absolute error is 10%), [23] (mean absolute error is 7.5%). An interesting observation is the declining simulation error seen when increasing the input data-set size of the QSort benchmark. As can be seen in Figure 4.2 the absolute error drops from 22% to 15% and 0.5% when increasing the data-set size from 500/1000 to 2000 and 5000 respectively. This is encouraging for running larger and more time-consuming benchmarks.

The simulation speedup achieved in our opinion outweighs the relatively small simulation error. As can be seen in Figure 4.1 the average simulation speedup is 15.19 with a geometric mean of 8.89 and harmonic mean of 7.87. Speedup ranges from 5.43 in MemLoadInd2048 up to 200.95 in MemStoreInd8192. In all cases the simulation time is significantly reduced when compared to RTL simulation.

To pin-point the main sources of error we employed more fine-grained performance measurements across the two models. We utilized microarchitectural counters in RSD and statistics counters in gem5 to measure memory accesses, cache misses, branch prediction misses, committed instructions and executed cycles. After gathering this data for all benchmarks we constructed graphs that represent the correlation of these fine-grained measurements with simulation error thus helping us find the system components that exhibit the highest mismatch. The aforementioned data can be observed in Figures 4.2 through 4.7.

After carefully analyzing the produced graphs we conclude that the memory system is the

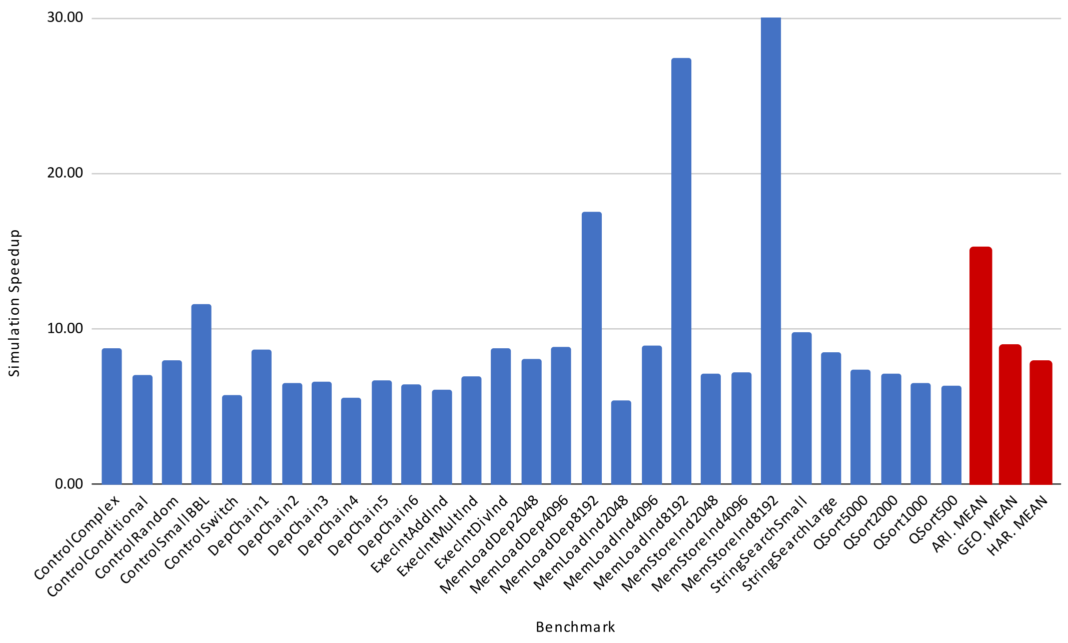


**Figure 4.1: Simulation Speedup gem5 vs RSD** [7]

main culprit behind the performance discrepancy between the two models. As seen in Figure 4.3 high memory accesses are correlated with simulation error. More specifically, as evidenced by Figures 4.4 and 4.5 which split memory accesses into loads and stores, a large quantity of loads favors RSD in speed whereas a large amount of stores favors gem5. In both cases the cycles ratio strongly deviates from the ideal value of 1.

Another source of error is the scheduler. In ExecIntAddInd we observed a drop in IPC only in RSD which led to 13% error between it and gem5. This is attributed to the fixed-priority scheduling scheme used in RSD that results in the pipeline stalling as the top priority and subsequent instructions are waiting to commit due to a lack of physical registers. The effect of this mismatch is much less felt than the mismatch of the memory system as for substantial error to occur all executed instructions need to have destination registers and the integer issue port has to be full in every cycle which is rare in real-world situations.

In conclusion, the average absolute error across all benchmarks is comparable to the results of other simulator validation studies. The main source of error is thought to be the memory system as evidenced by the low average absolute error of Control, Dependency and Execution (not including ExecIntDivInd) microbenchmarks i.e 5.89%, 1.53% and 7.47% respectively when compared to the high absolute error of Memory microbenchmarks which amounts to 36.53%.

[7]The simulation speedup in MemStoreInd8192 is 200.95. For scaling purposes the graph cuts off at 30

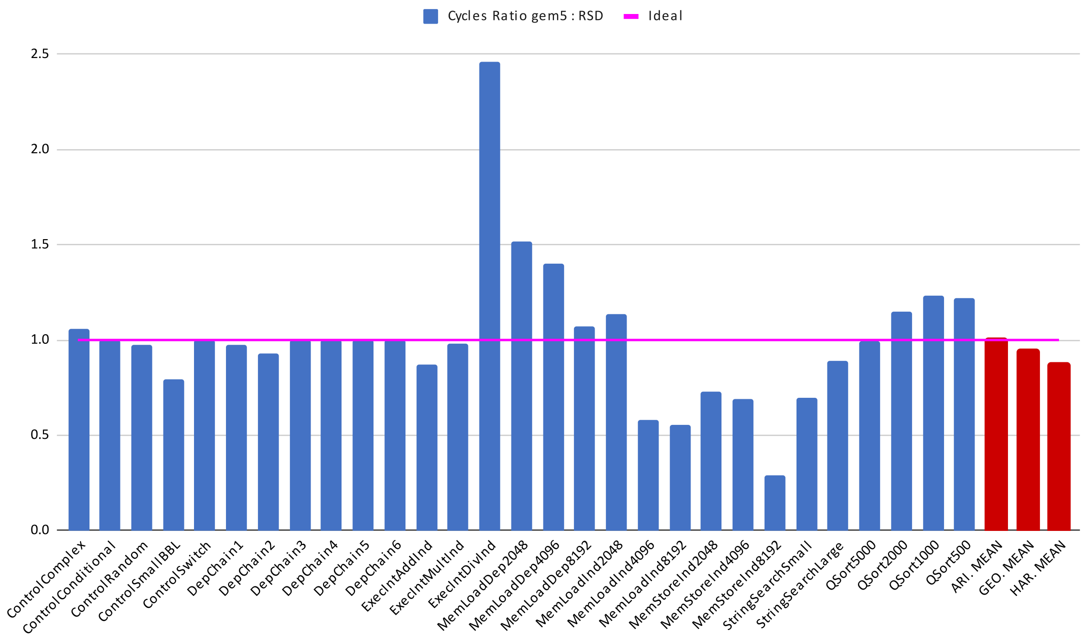


**Figure 4.2: Cycles Ratio** $gem5 : RSD$

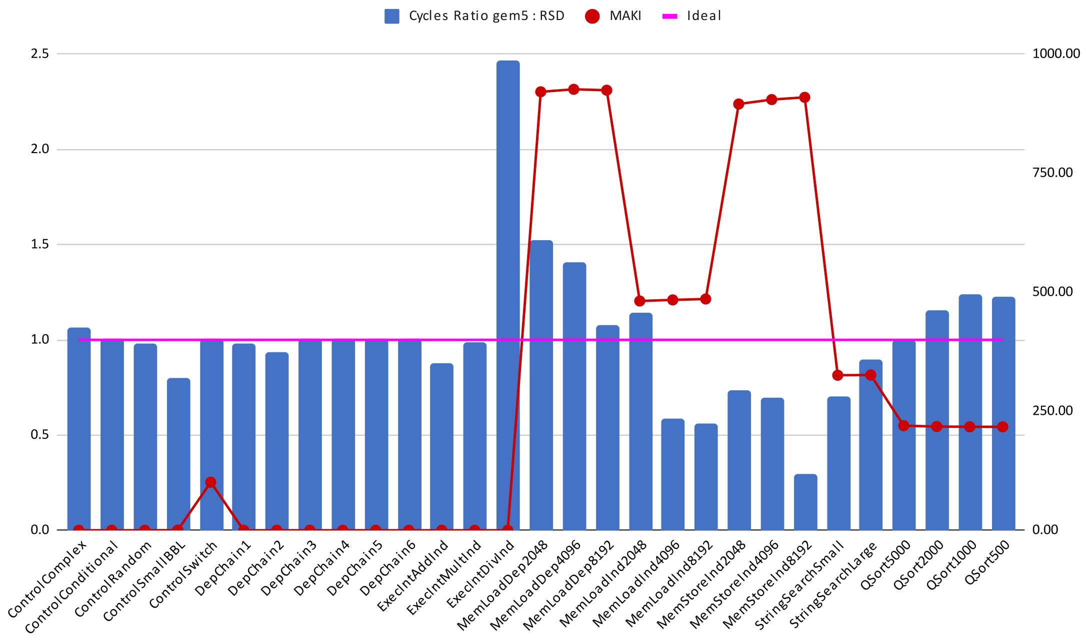


**Figure 4.3: Memory Accesses per Kilo-Instructions vs Cycles Ratio** $gem5 : RSD$

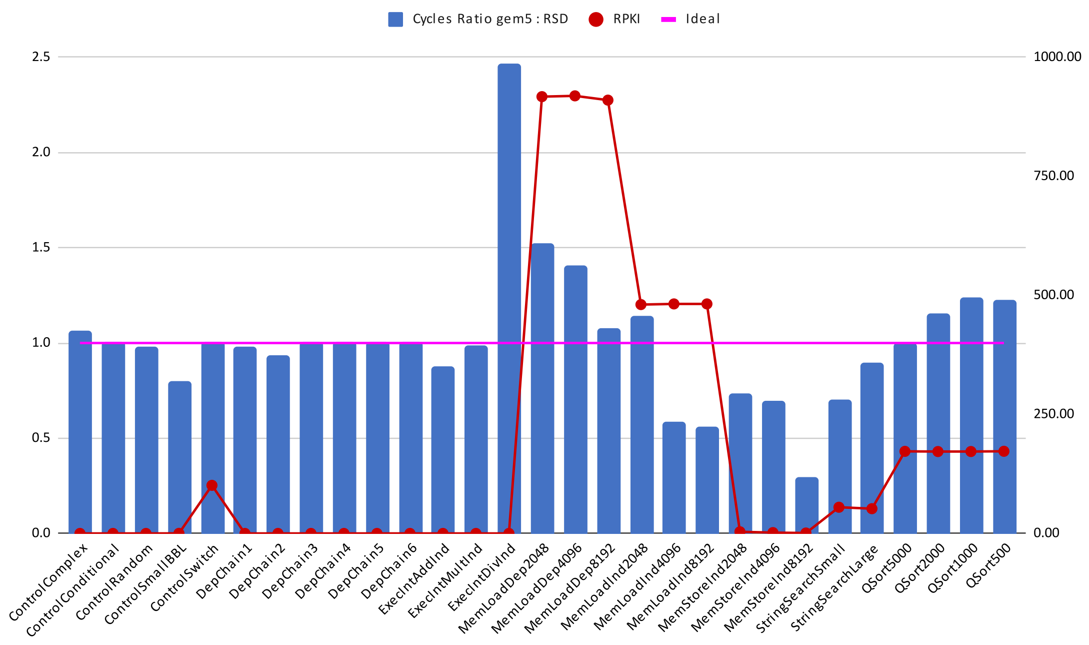


**Figure 4.4: Memory Reads per Kilo-Instructions vs Cycles Ratio** $gem5 : RSD$

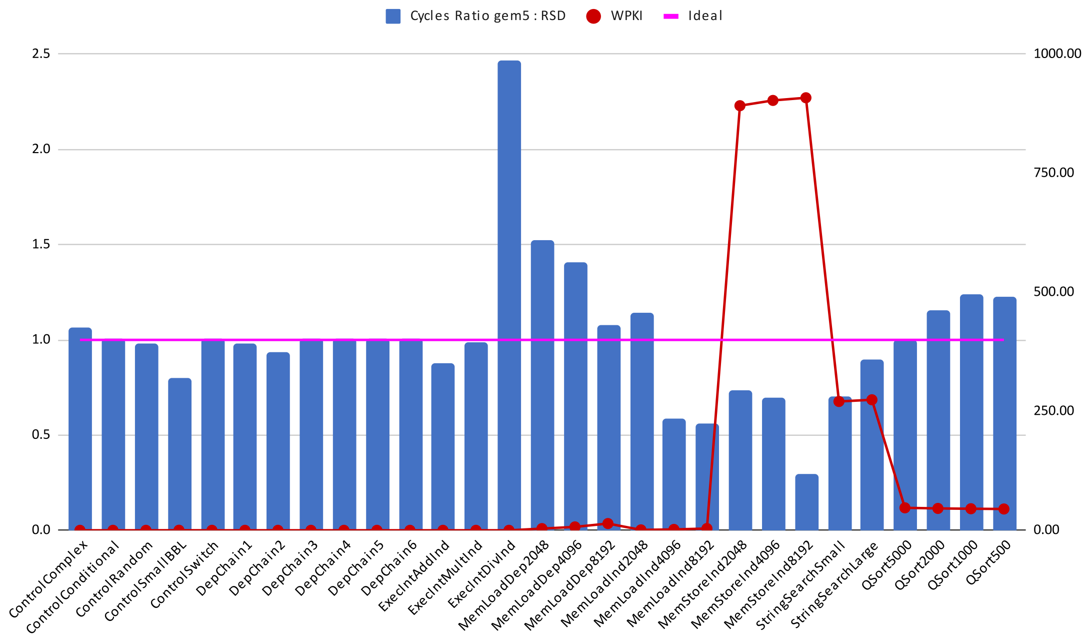


**Figure 4.5: Memory Writes per Kilo-Instructions vs Cycles Ratio** $gem5 : RSD$

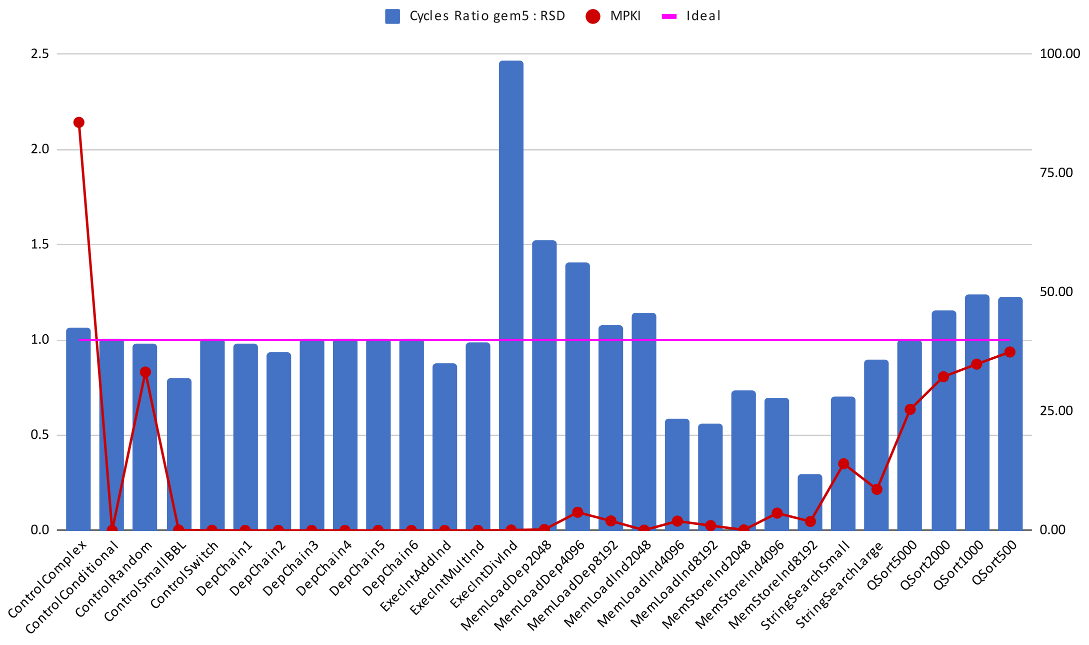


**Figure 4.6: Branch Prediction Misses per Kilo-Instructions vs Cycles Ratio** $gem5 : RSD$

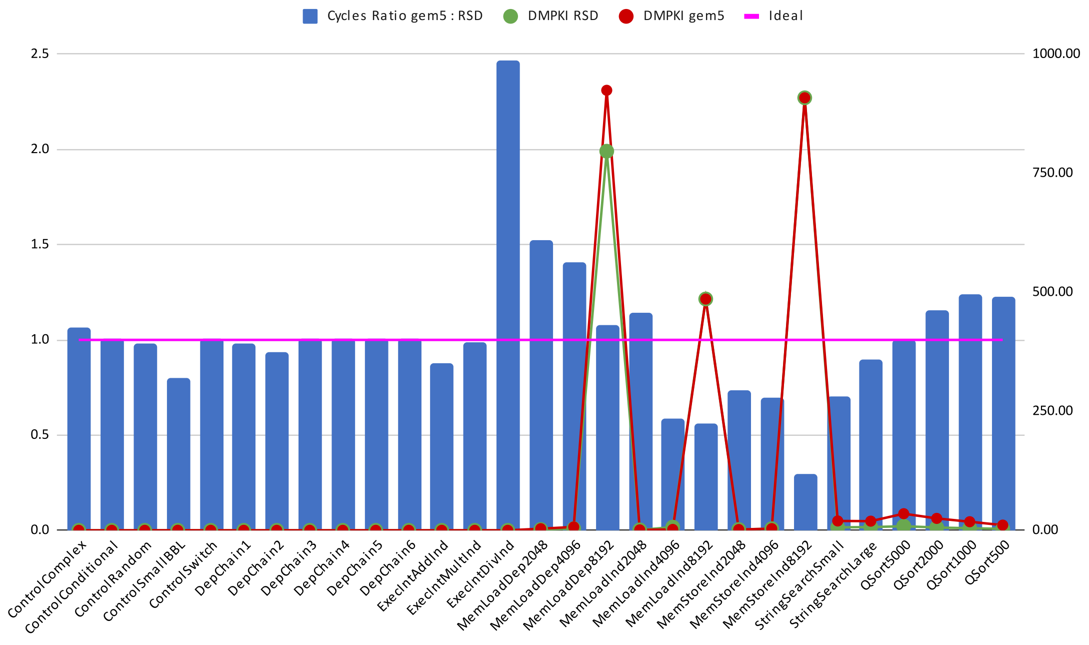


**Figure 4.7: D$ Misses per Kilo-Instructions vs Cycles Ratio** $gem5 : RSD$

# 5. FUTURE WORK

As a continuation of our effort to match gem5's to RSD's performance we aim to implement new features and make changes to our already existing work. Our goals in the future are to streamline the microarchitectural parameter discovery and improve the accuracy of performance simulation [20].

In order to streamline microarchitectural parameter discovery we seek to design a framework of scripts that can stress different parts of the microprocessor and determine the sizes of different components as well as the type and amount of functional units. Aside from this we have considered using a machine learning model to automate the parameter discovery and optimization process by providing it with fine grained measurements of performance (i.e microarchitectural counters) of a vast array of test programs.

In the simulation accuracy front since we have pinpointed the main source of error to be the memory system we aim to make targeted changes to it. One potential culprit seems to be the MemoryLatencySimulator of RSD which seems to handle accesses to main memory in a different manner than gem5. After careful analysis of RSD's implementation we hope to be able to modify gem5 to more closely match the behavior of RSD.

Departing from our RTL vs microarchitecture level simulation theme we also plan to use more advanced microprocessor models such as that of BOOM in order to validate the Full System simulation mode of gem5 this time against a hardware simulation of an RTL model using the FireSim framework. This will give us a access to a full fledged Linux capable system that will make benchmarking with much more advanced programs and frameworks easier. Also simulation speed is expected to increase with the FPGA acceleration providing us with a much more competitive framework when compared to microarchitecture-level simulators.

We look forward to addressing all the above research topics in the future and believe that model validation of RISC-V processors and systems will be a fruitful research area for many years as the popularity of RISC-V and open hardware design become increasingly popular.

# ABBREVIATIONS - ACRONYMS

| | |
|---|---|
| SoC | System on Chip |
| ISA | Instruction Set Architecture |
| RISC | Reduced Instruction Set Computer |
| IPC | Instructions Per Cycle |
| FSM | Finite State Machine |
| OoO/O3 | Out of Order |
| ABI | Application Binary Interface |
| RTL | Register Transfer Level |
| MIPS | Million Instructions Per Second |
| ILP | Instruction Level Parallelism |
| IQ | Issue Queue |
| ROB | Reorder Buffer |
| RMT | Register Map Table |
| LDQ | Load Queue |
| STQ | Store Queue |
| PRF | Physical Register File |
| LSU | Load Store Unit |
| CAM | Content Addressable Memory |
| MDP | Memory Dependence Predictor |
| PHT | Pattern History Table |
| BTB | Branch Target Buffer |
| MSHR | Miss Status Holding Register |
| SE | System-call Emulation mode |
| FS | Full System mode |
| IEW | Issue/Execute/Writeback |
| LFSR | Linear Feedback Shift Register |

# REFERENCES

[1] Apple Inc.

[2] The gem5 simulator.

[3] MARSSx86 - Micro-ARchitectural and System Simulator for x86-based Systems.

[4] OVPsim Simulator | Open Virtual Platforms.

[5] RISC-V International.

[6] The RISC-V ISA Simulator (Spike).

[7] The Sniper Multi-Core Simulator.

[8] Specifications - RISC-V International, Sep 2021.

[9] Daniel Aarno and Jakob Engblom. *Software and System Development Using Virtual Platforms: Full-system Simulation with Wind River Simics*. Morgan Kaufmann, 2014.

[10] Ayaz Akram and Lina Sawalha. *Validation of the gem5 Simulator for x86 Architectures*. pages 53–58. IEEE, 2019.

[11] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. *Sinuca: A validated micro-architecture simulator*. pages 605–610. IEEE, 2015.

[12] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. *Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs*, 2020.

[13] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. *The Rocket Chip Generator*. 2016.

[14] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[15] T. Austin, E. Larson, and D. Ernst. *SimpleScalar: an infrastructure for computer system modeling*. *Computer*, 2002.

[16] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. *Chisel: Constructing Hardware in a Scala Embedded Language*, 2012.

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. *The gem5 Simulator*. page 1–7, 2011.

[18] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. *Accuracy Evaluation of gem5 Simulator System*, 2012.

[19] Juan M Cebrian, Adrián Barredo, Helena Caminal, Miquel Moretó, Marc Casas, and Mateo Valero. *Semi-automatic validation of Cycle-accurate Simulation Infrastructures: The Case for gem5-x86*. pages 832–847, 2020.

[20] Odysseas Chatzopoulos, George-Marios Fragkoulis, George Papadimitriou, and Dimitris Gizopoulos. *Towards Accurate Performance Modeling of RISC-V Designs*. 2021.

[21] Dr. Ian Cutress and Andrei Frumusanu. *AMD Zen 3 RYZEN deep Dive REVIEW: 5950X, 5900X, 5800x and 5600X Tested*, 2020.

[22] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. *A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations*. page 12–20, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. *Micro-architectural Simulation of In-Order and Out-of-Order ARM Microprocessors with gem5*. pages 266–273. IEEE, 2014.

[24] Federico Faggin. *Silicon: From the Invention of the Microprocessor to the New Science of Consciousness*. Waterside Productions, 2021.

[25] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. *Processor Microarchitecture: An Implementation Perspective*. Morgan & Claypool Publishers, 2010.

[26] Timothy H. Heil, Zak Smith, and J. E. Smith. *Improving Branch Predictors by Correlating on Data Values*. MICRO 32, page 28–37. IEEE Computer Society, 1999.

[27] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. *Sniper: Scalable and accurate parallel multi-core simulation*. pages 91–94, 2012.

[28] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2019.

[29] John L Hennessy and David A Patterson. *A New Golden Age for Computer Architecture*. pages 48–60, 2019.

[30] Alexey Lesnykh. *Computer Simulation: Basics, Terminology, Levels*, 2020.

[31] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. *The gem5 Simulator: Version 20.0+*. 2020.

[32] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. *Simics: A full system simulation platform*. 2002.

[33] Susumu Mashimo, Akifumi Fujita, Reoma Matsuo, Seiya Akaki, Akifumi Fukuda, Toru Koizumi, Junichiro Kadomoto, Hidetsugu Irie, Masahiro Goshima, Koji Inoue, and Ryota Shioya. *An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor*. pages 63–71, 2019.

[34] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. *MARSS: A full system simulator for multicore x86 CPUs*. pages 1050–1055, 2011.

[35] David A Patterson and John L Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*.

[36] QawsQAER. *gem5 gshare Branch Predictor*.

[37] Sheng-bing REN, Wan-li ZHANG, Nian LU, and Zhen-yu PAN. *Education Research on Multi-core Simulation of Embedded System Based on OVPsim [J]*. 2010.

[38] Ryota Shioya. *Konata Pipeline Visualizer*.

[39] Leena Singh and Leonard Drucker. *Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout*. Springer Science & Business Media, 2007.

[40] Brian Slechta, David Crowe, N Fahs, Michael Fertig, Gregory Muthler, Justin Quek, Francesco Spadini, Sanjay J Patel, and Steven S Lumetta. *Dynamic Optimization of Micro-Operations*. pages 165–176. IEEE, 2003.

[41] Wilson Snyder. *Verilator, Accelerated: Accelerating development, and case study of accelerating performance*. 2020.

[42] Tze Sin Tan and Bakhtiar Affendi Rosdi. *Verilog HDL simulator technology: a survey*. pages 255–269, 2014.

[43] Stephen Williams and Michael Baxter. *Icarus Verilog: Open-Source Verilog More Than A Year Later*. 2002.

[44] Matt T. Yourst. *PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator*. pages 23–34, 2007.

[45] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.