Bachelor Thesis

Benchmark of RISC-V in BTOR2

Jan Krister Möller

Examiner: Dr. Mathias Fleury

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Computer Architecture

August 19, 2025

Writing Period

24. 06. 2025 – 24. 09. 2025

Examiner

Dr. Mathias Fleury

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Declaration on Usage of generative AI

I hereby declare that, with the approval of my examiner, I have employed the generative AI tool "GitHub Copilot" during the preparation of this thesis solely for spellchecking and enhancing the formality of my written expression. Furthermore, I expressly confirm that this tool was not used to generate data or as a source of factual information or content for this thesis.

Place, Date

Signature

Abstract

foo bar [1] [2] [3]

Contents

List of Figures

List of Tables

List of Algorithms

1 Motivation

This is a template for an undergraduate or master's thesis. The first sections are concerned with the template itself. If this is your first thesis, consider reading.

2 RISC-V

As the first foundation for my benchmarks and, consequently, this thesis, I will discuss RISC-V and its operational principles.

2.1 Overview

RISC-V is an open-source instruction set architecture first published in May 2011 by A. Waterman et al. [4]. As indicated by its name, it is based on the RISC design philosophy. **(TODO: Explain RISC (compare wiki))** Since 2015, the development of RISC-V has been coordinated by the RISC-V International Association, a non-profit corporation based in Switzerland since 2020 [5]. Its objectives include providing an *open* ISA that is freely available to all, a *real* ISA suitable for native hardware implementation, and an ISA divided into a *small* base integer ISA usable independently, for example in educational contexts, with optional standard extensions to support general-purpose software development [1, Chapter 1].

Currently, RISC-V comprises four base ISAs: RV32I, RV64I, RV32E, and RV64E, which can be extended with one or more of the 47 ratified extension ISAs [1, Preface].

(EXTEND: Additional content may be required here) **(TODO: Mention little endian?)**

For the purposes of this work, I will focus on a subset of the RV64I ISA.

2.2 The RV64I ISA

RV64I is not overly complex, but its structure is essential for understanding the subsequent work presented in this thesis. Therefore, I will explain all elements relevant to my research.

RV64I features 32 64-bit registers, labeled $x0$–$x31$, where $x0$ is hardwired to zero across all bits. Registers $x1$–$x31$ are general-purpose and may be interpreted by various instructions as collections of booleans, two's complement signed binary integers, or unsigned integers. Additionally, there is a register called *pc*, which serves as the program counter and holds the address of the current instruction [1, Chapters 4.1, 2.1].

In RV64I, memory addresses are 64 bits in size. As the memory model is defined to be single-byte addressable, the address space of RV64I encompasses 2^{64} bytes [1, Chapter 1.4].

Like nearly all standard ISAs of RISC-V, RV64I employs a standard instruction encoding length of 32 bits, or one *word*. Only the compressed extension C introduces instructions with a length of 16 bits [1, Chapter 1.5], which is not relevant for this discussion. All RV64I instructions are encoded in one of the six formats illustrated in Figure 1.

The design of these formats results in the following features:

- Due to RISC-V's little-endian nature, the *opcode*, which encodes the general instruction, is always read first. Further specification of the instruction via $funct3$ and $funct7$ is consistently located at the same positions.

- If utilized by the instruction, the destination register rd and the source registers $rs1$ and $rs2$ are always found in the same locations, simplifying decoding.

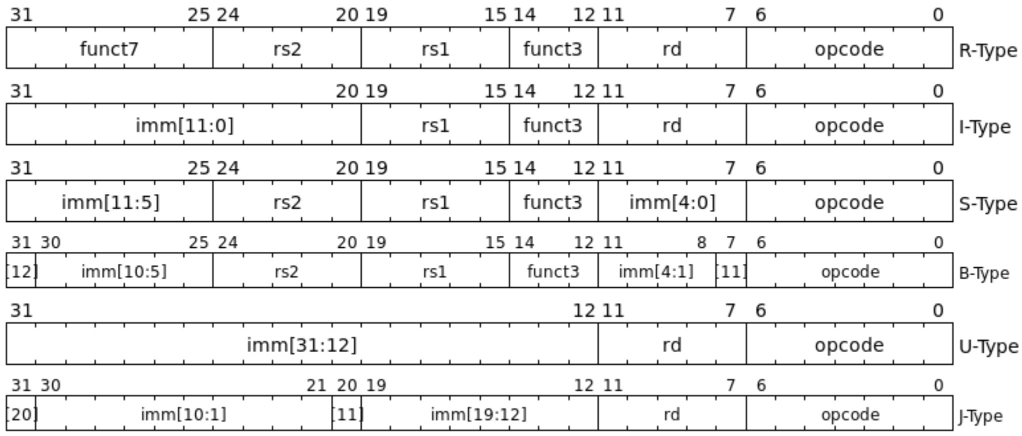

Figure 1: RV64I encoding formats, used in [1, Chapter 2.3]

- The highest bit of the immediate value *imm* is always bit 31, making it straightforward to sign-extend the immediate value.

Note that each immediate subfield is labeled with its bit position within the immediate value. Immediate values are always sign-extended to 31 bits, and in the case of U-, B-, and J-type formats, the missing lower bits are filled with zeros.

The instructions relevant to my work are listed in Table 1

I have divided the instructions in Table 1 into nine groups based on their operations. `LUI` and `AUIPC` move a high immediate into *rd*; `JA*` instructions are unconditional jumps, and `B*` instructions are conditional jumps. `L*` instructions load sign-extended values from memory, either as `B`yte, `H`alfword, `W`ord, or `D`oubleword lengths. Conversely, `S*` instructions write values of the specified length to memory. **(TODO: arithmetic)** Note that the suffix `U` denotes operations where values are processed as unsigned.

I left out `FENCE`, `ECALL` and `EBREAK` instructions as without I/O interaction or an environment like an OS or a debugger, these are not needed.

INSTR	TYPE	INSTR	TYPE	INSTR	TYPE	INSTR	TYPE
LUI	U	LW	I	XORI	I	SLT	I
AUIPC	U	LD	I	ORI	I	SLTU	I
JAL	J	LBU	I	ANDI	I	XOR	I
JALR	I	LHU	I	SLLI	I	OR	I
BEQ	B	LWU	I	SRLI	I	AND	I
BNE	B	SB	S	SRAI	I	SLL	I
BLT	B	SH	S	ADDIW	I	SRL	I
BGE	B	SW	S	SLLIW	I	SRA	I
BLTU	B	SD	S	SRLIW	I	ADDW	I
BGEU	B	ADDI	I	SRAIW	I	SLLW	I
LB	I	SLTI	I	ADD	I	SRLW	I
LH	I	SLTIU	I	SUB	I	SRAW	I

Table 1: Subset of RV64I instructions **(TODO: Maybe rework, not happy yet)**

2.3 Simulation of RISC-V

(TODO: This may be better placed in Chapter 4, but the state file is relevant here.)

2.3.1 Saving the State of a RISC-V Processor

To preserve the current state of a RISC-V processor, both the registers and memory must be stored. For this purpose, I have devised the format shown in Figure 2. The minimal file consists only of the two designators "REGISTERS:" and "MEMORY:", the current *pc*, and one empty line.

REGISTERS:
PC: *current pc in hex*
x*(0 - 31)*: *value of register in hex*

MEMORY:
(address in hex): *byte, halfword, word or doubleword in hex*

Figure 2: Construction of .state files

3 BTOR2

The second foundation of my benchmarks is BTOR2, a word-level model checking format published by A. Niemetz et al. [2].

3.1 Model Checking

(TODO: Write something about model checking...)

3.2 The BTOR2 Language

Generally in BTOR2, every line represents either a sort or a node, where normally the line number acts as an identifier. A sort behaves similar to a type as with it, either the length of a bitvector or the size of an array of bitvectors is defined. Nodes on the other hand represent a value of a defined sort and come as constants, operations or constraints. These values can later on be referenced by the node identifier, so the line number. The syntax of BTOR2 can be found at [2, figure 1] and corresponding operators in [2, table 1]

Key features of BTOR2 include its ability to operate sequentially, which makes the implementation of a RISC-V structure highly convenient. The main feature is the `state` operator, which defines a node that is sequentially updated. With an `init` node, this state can be assigned an initial value, and with `next`, the sequentially next state can be defined. Finally, constraints can be used to specify endpoints for a model. These endpoints may indicate that something unintended has occurred or that the intended information has been found. In either case, the resulting model is provided as a witness.

3.3 The BTOR2 Witness

After receiving a witness, it must be interpreted. On the second line of a witness, the constraint that was triggered is specified. Subsequently, for each sequential iteration, the witness first presents—marked with `#`x, where x is the iteration number—a representation of all states in the current iteration. Second, marked with `@`x, all inputs for the iteration are listed.

4 Transforming RISC-V to BTOR2

4.1 The Concept

4.2 Encoding

4.2.1 Constants

4.2.2 State Representation

4.2.3 Initialization

4.2.4 Computing values

Opcode

funct3 & funct7

Registers

Immediate

4.2.5 Command Detection

4.2.6 Next-State-Logic

4.2.7 Constraints

4.3 Testing for Correctness

4.3.1 State Fuzzer

4.3.2 Automated Logging

4.4 Functional vs Relational Next-State-Logic

5 Benchmarks

5.1 MultiAdd in Functional and Relational Next-State-Logic

5.2 Memory Operations

5.3 Results

Bibliography

[1] *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, 2025, version 20250508. [Online]. Available: https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications

[2] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , BtorMC and Boolector 3.0," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 587–595.

[3] F. Schrögendorfer, "Bounded Model Checking of Lockless Programs," Master's thesis, Johannes Kepler University Linz, August 2021. [Online]. Available: https://epub.jku.at/obvulihs/download/pdf/6579523

[4] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: Base user-level isa," UC Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html

[5] "History of RISC-V," https://riscv.org/about/, accessed: 15.08.2025.