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

September 12, 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)

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 work.

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 non-accessible 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]. The format of the memory is little endian, so the lower bits of number are placed on lower addresses.

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 named C introduces instructions with a length of 16 bits [1, Chapter 1.5], but we will not encounter this special case. All RV64I instructions are encoded in one of the six formats illustrated in Figure 1. These formats may consist of

- The *opcode*:
  The opcode is used to differ between groups of instructions. It also defines the format type of the instruction.

- *rd*:
  This is the destination register.

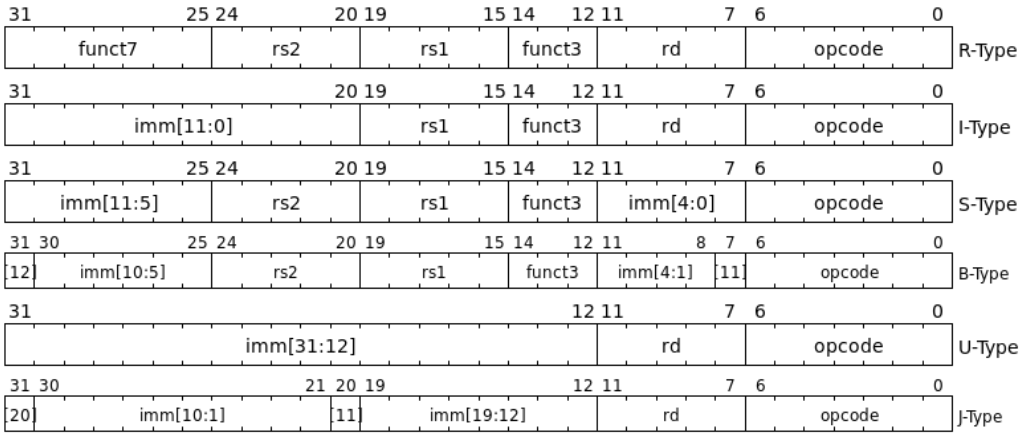

Figure 1: RV64I encoding formats, used in [1, Chapter 2.3]

- $funct3$:

  This is used to differ between instructions with the same *opcode*.

- $rs1$ & $rs2$:

  These are the source registers.

- $funct7$

  :This is used for further distinctions between instructions if there are more than 8 instructions in an opcode group and $funct3$ does not suffice.

- imm:

  This is an immediate value. In square brackets after imm is designated a subfield of the immediate which is represented by these bits. From these subfields, non-defined lower bits are filled with zeros whereas the highest defined bit is sign extended to fill all non-defined higher bits.

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.

Instr	*opcode*	Type
LUI	*lui*	U
AUIPC	*auipc*	
JAL	*jal*	J
JALR	*jalr*	I
BEQ	*branch*	B
BNE		
BLT		
BGE		
BLTU		
BGEU		
LB	*load*	I
LH		
LW		
LD		
LBU		
LHU		
LWU		

Instr	*opcode*	Type
SB	*store*	S
SH		
SW		
SD		
ADDI	*op-imm*	I
SLTI		
SLTIU		
XORI		
ORI		
ANDI		
SLLI		I*
SRLI		
SRAI		
ADDIW	*op-imm-32*	I
SLLIW		I**
SRLIW		
SRAIW		

Instr	*opcode*	Type
ADD	*op*	R
SUB		
SLT		
SLTU		
XOR		
OR		
AND		
SLL		
SRL		
SRA		
ADDW	*op-32*	R
SUBW		
SLLW		
SRLW		
SRAW		

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

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

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

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. In case of `AUIPC`, the pc is added onto this.

`JAL` and `JALR` instructions are unconditional jumps, where on `JAL` imm is added onto pc and on `JALR` imm is added onto $rs1$ and set as pc. Both link to the next instruction (current $pc + 4$) in rd.

Instr	Behavior
ADD	$rd := rs1 + rs2$
SUB	$rd := rs1 - rs2$
SLT	$rd := 1$ if $rs1 < rs2$ else $rd := 0$
XOR	$rd := rs1 \oplus rs2$, bitwise
OR	$rd := rs1 \vee rs2$, bitwise
AND	$rd := rs1 \wedge rs2$, bitwise
SLL	$rd := rs1$ shifted left by $rs2$, new bits are zeros
SRL	$rd := rs1$ shifted right by $rs2$, new bits are zeros
SRA	$rd := rs1$ shifted right by $rs2$, sign extend

Table 2: All suffix free operations in RV64I and their behavior. All values are handled either bitwise or as signed twos complement integers

branch instructions are conditional jumps. $rs1$ is compared to $rs2$ and if the comparison holds, *imm* is added onto *pc*. The comparison are $=$ for BEQ, $\neq$ for BNE, $<$ for BLT and $\geq$ for BGE. In these instructions, the values in $rs1$ and $rs2$ are handled as twos complement integers. The suffix *U in an instruction generally designates an unsigned operation. In this case the values in $rs1$ and $rs2$ are handled as unsigned integers. Apart from this, they work as their counterpart without the suffix.

load instructions load values from memory at address $(rs1 + imm)$ into rd, either at Byte, Halfword, Word, or Doubleword length. Again the standard is a sign-extended value and the suffix *U designates the loading of a non sign extended value.

Conversely, *store* instructions write values from $rs2$ at the address $(rs1 + imm)$ to memory. Here also the distinction between the different lengths is made and the lowest byte, halfword, word or the whole doubleword is stored at the address.

All further instructions can be seen as generic operations, differentiated by their suffixes. To simplify the explanation process, all operations without any suffix and their behavior are listed in Table 2. This is almost exactly the group with opcode *op*, except the SLTU instruction, which is not suffix free. But as all other instructions with the unsigned suffix it behaves as its signed counterpart except from handling both $rs1$ and $rs2$ as unsigned integers.

These operations can be extended by the *I suffix which is designated by the opcode $op-imm$. This exchanges $rs2$ with imm in the behavior. Again, SLTI can be extended to an unsigned version SLTIU, which behaves as expected. A SUBI instruction does not exist as it is redundant. Its behavior can be reached by using ADDI with a negative immediate.

Also, the operations ADD, SUB, SLL, SRL and SRA can be extended with the *W suffix. This forms the group with the opcode $op - 32$. In contrast to the base instructions these new ones behave as if the registers are only 32bit. The result is placed in the low 32 bits of rd and sign extended to full 64bit. Overflows are ignored.

The last group left is the combination of both suffixes *IW with the opcode $op-imm-32$. The behavior differs from the base instructions, as expected, by a replacement of $rs2$ with imm and only operating on 32bit. Again, a SUBIW instruction is redundant as a negative immediate with ADDIW archives the same.

Compared to the full RV64I ISA, 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.

2.3 Simulation of RISC-V

In a previous project, I have written a simulation of this subset of RV64I in C [6]. As I will use it to later on test my BTOR2 model, I explain the inner working of the program here.

First I implemented a structure to represent a simple RISC-V processor

```c
typedef struct memory_cell
{
    uint64_t address;
    uint8_t content;
    struct memory_cell *next_cell;
} memory_cell;
```

```c
typedef struct memory_table
{
    memory_cell *memory[TABLESIZE];
    uint64_t initialised_cells;
} memory_table;
```

```c
typedef struct state
{
    uint64_t pc;
    uint64_t regs_values[32];
    bool regs_init[32];
    memory_table *memory;
} state;
```

Figure 2: State representation of a RISC-V processor in the simulation [6]

2.3.1 Representing the State of a RISC-V Processor

Of course the state needs a representation for all registers. *pc* is defined as a 64bit integer, the other 32 registers are implemented as an array, as so each register can be referenced by its number. Also, I implemented an array of flags, one for each register, to be able to differ between initialized and non initialized registers. The memory is build from single memory cells holding each an address and its byte of content. These are cumulated in a hash table "memorytable", hashing on the address. If adding a new cell causes a collision, it is appended to the cells already in the bucket forming a linked list. These structs are shown in Figure 2.

2.3.2 Running an instruction

After fetching the current instruction from the hash table, it must be decoded. The easiest way to do this is using a decision tree. First I mask out the opcode and match it over all implemented opcodes. From there, either this is an endpoint and the instruction is identified, or $funct3$ must be masked and matched. A final differentiation over $funct7$ might be needed, but after this every Leaf in the tree

coincides with an instruction. So, at every leaf the state can be changed in regard to the corresponding instruction. **(TODO: Graph of the decision tree?)**

2.3.3 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 3. The RISC-V simulation uses this format as input and output. The minimal file consists only of the two designators "REGISTERS:" and "MEMORY:" and one empty line between them. Under "REGISTERS:", all registers can be listed with their corresponding value. Of course, x0 can not be different from 0. I included the option to reference it nonetheless to have the complete state included. Under "MEMORY:", after giving an address, the memory can be filled with 1-, 2-, 4- or 8-byte sized memory content. The given address is the starting address of the content, of course with every byte after the first, the next higher address is filled.

$\langle 64bitHex \rangle$	::=	up to 16 digits of [0-9a-fA-F]
$\langle 32bitHex \rangle$	::=	up to 8 digits of [0-9a-fA-F]
$\langle 16bitHex \rangle$	::=	up to 4 digits of [0-9a-fA-F]
$\langle 8bitHex \rangle$	::=	up to 2 digits of [0-9a-fA-F]
$\langle memContent \rangle$	::=	$\langle 8bitHex \rangle$
		| $\langle 16bitHex \rangle$
		| $\langle 32bitHex \rangle$
		| $\langle 64bitHex \rangle$
$\langle cell \rangle$	::=	$\langle 64bitHex \rangle$`:`$\langle memContent \rangle$
		| $\langle cell \rangle \langle cell \rangle$
$\langle regNum \rangle$	::=	`0 |...| 31`
$\langle reg \rangle$	::=	`PC:`$\langle 64bitHex \rangle$`\n`
		| `x`$\langle regNum \rangle$`:`$\langle 64bitHex \rangle$`\n`
		| $\langle reg \rangle \langle reg \rangle$
$\langle memory \rangle$	::=	`MEMORY:\n`$\langle cell \rangle$
$\langle registers \rangle$	::=	`REGISTERS:\n`$\langle reg \rangle$
$\langle state \rangle$	::=	$\langle registers \rangle$`\n`$\langle memory \rangle$`\n`

Figure 3: 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 a `next` node, 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.

(TODO: Maybe a bit more, it's a bit of bare bones)

4 Transforming RISC-V to BTOR2

(TODO: Explain naming conventions for the model nodes)

This chapter addresses the main problem of the thesis: transforming a RISC-V state into the BTOR2 format for benchmarking purposes. F. Schrögendorfer did similar in his master's thesis "Bounded Model Checking in Lockless Programs" [3], in which he describes, among other topics, an encoding concept for a minimal machine in a multiprocessor context [3, Chapter 2]. For this, in [3, Chapter 8] he describes a way to encode programs for his machine model into a BTOR2 model. This can not be replicated by me though, as in his model the full program is known at encoding whilst I want to hold the property of RISC-V that the program could self modify and change during execution. If the property was to be dropped, it would be possible to parse over the memory and analyse the complete behavior of a program within, but this is for others to explore.

4.1 The Concept

To successfully execute a RISC-V instruction, three fundamental steps must occur in sequence:

- Fetch the current instruction from memory
- Identify the instruction

- Execute the instruction

Due to the fixed instruction length of RISC-V, as mentioned in Section 2.2, fetching the current instruction is straightforward. Ultimately, we want a node that retrieves a *word* from memory at the location specified by *pc*.

For basic identification, the *opcode* must be extracted and checked. Depending on the opcode, further distinctions between instructions require extracting and checking $funct3$ and, if necessary, $funct7$. Ultimately, we want a node for each instruction, which holds a boolean value indicating whether this instruction was fetched.

To execute the instruction, we need to extract the values of the immediate imm and, if used, the registers $rs1$ and $rs2$. All instructions only modify rd, *pc*, or memory. Therefore, the next-state logic can be generalized for these three cases.

Memory is only modified when a store instruction is identified. As all store instructions share the same type, computing the memory address is consistent across them. The final step is overwriting the memory at this address.

For the *pc*, except for jump commands, it always increments to point to the next instruction. The two unconditional jumps, `JAL` and `JALR`, must be handled separately. For branch instructions, after determining whether the relevant condition for the instruction holds, we can generalize, as all branch instructions execute the same operation from this point onward.

With rd, generalization across instructions is not feasible. However, we can generalize across all possible registers by adding a check in each register's update function to determine whether the register in question is rd.

4.2 Encoding

For better visualization in the BTOR2 code I will mark all sort-IDs in gray, all node-IDs in red and all non-ID numbers blue. As described in the BTOR2 syntax [2, Figure 1], each line can get an accompanying symbol. Sadly those cant be used as an alias to the line numbers, but for increased clarity, in the following figures I will use them as such aliases. With this I can also start each new figure with the relative line number `n`, and it makes it feasible to describe processes with algorithms. It is implied that `n` is sufficiently incremented after adding to the model so that IDs will not overlap. In the following, I will describe how I construct a BTOR2 model for a RISC-V state file.

4.2.1 Constants

First off, I added the sorts and non-progressive constants needed into the BTOR2 model as seen in Figure 4. This is extended by a set of progressive constants used for comparison e.g. against the register number. Algorithm 1 describes how they are added.

Of note is the Representation of the memory as an array of addressable memory cells of each 1byte. Obviously, the set address space of 16bit is magnitudes away of the expected address space of 64bit, but representing a 64bit addressable memory with its resulting $2^{64}B \approx 18Exabyte$ is not implementable. Therefore, as I needed a feasible amount of memory space, I artificially chose a 16bit address space as a soft minimum. With $65kB$ and therefore programs with possibly > 10000 instructions I deemed this memory sufficient for most use cases. Despite this, the encoding is implemented in such a way that the address space can be altered with. **(TODO: Change code to make address space modifiable by an option)**

```
1  sort    bitvec  1          Bool
2  sort    bitvec  16         AS
3  sort    bitvec  8          B
4  sort    bitvec  16         H
5  sort    bitvec  32         W
6  sort    bitvec  64         D
7  sort    array   2       3  Mem

8  one     Bool               true
9  zero    Bool               false
10 one     AS                 addressInc
11 constd  AS      4          pcInc
12 zero    B                  emptyCell
13 one     W                  bitPicker
14 zero    D                  emptyReg

15 consth  W       01F        5Bitmask
16 consth  W       03F        6Bitmask
17 consth  W       07F        7Bitmask
18 consth  W       0FFF       12Bitmask
19 consth  W       0FFFFF     20Bitmask

20 constd  W       7          shiftToRd
21 constd  W       15         shiftToRs1
22 constd  W       20         shiftToRs2
23 constd  W       12         shiftToFunct3
24 constd  W       25         shiftToFunct7
25 constd  W       5          shiftBy5
26 constd  W       11         shiftBy11

27 constd  W       3          load
28 constd  W       19         opImm
29 constd  W       23         auipc
30 constd  W       27         opImm32
31 constd  W       35         store
32 constd  W       51         op
33 constd  W       55         lui
34 constd  W       59         op32
35 constd  W       99         branch
36 constd  W       103        jalr
37 constd  W       111        jal
```

(TODO: Maybe neusortieren, andere constanten aufnehmen. Explain)

Figure 4: Sorts and non-progressive Constants for encoding RISC-V in BTOR2

for i from 0 to 31 **do**

add to model:

n	constd	W	i	*iConst*

end

Algorithm 1: progressive constants for encoding RISC-V in BTOR2

(n + 0)	state	D	*x0*
(n + 1)	state	D	*x1*
(n + 2)	state	D	*x2*
(n + 3)	state	D	*x3*
(n + 4)	state	D	*x4*
(n + 5)	state	D	*x5*
(n + 6)	state	D	*x6*
(n + 7)	state	D	*x7*
(n + 8)	state	D	*x8*
(n + 9)	state	D	*x9*
(n + 10)	state	D	*x10*
(n + 11)	state	D	*x11*
(n + 12)	state	D	*x12*
(n + 13)	state	D	*x13*
(n + 14)	state	D	*x14*
(n + 15)	state	D	*x15*
(n + 16)	state	D	*x16*
(n + 17)	state	D	*x17*
(n + 18)	state	D	*x18*
(n + 19)	state	D	*x19*
(n + 20)	state	D	*x20*
(n + 21)	state	D	*x21*
(n + 22)	state	D	*x22*
(n + 23)	state	D	*x23*
(n + 24)	state	D	*x24*
(n + 25)	state	D	*x25*
(n + 26)	state	D	*x26*
(n + 27)	state	D	*x27*
(n + 28)	state	D	*x28*
(n + 29)	state	D	*x29*
(n + 30)	state	D	*x30*
(n + 31)	state	D	*x31*
(n + 32)	state	AS	*pc*
(n + 33)	state	Mem	*memory*

Figure 5: State representation for encoding

4.2.2 State Representation

The next logical step is defining a representation of a RISC-V state. This is straightforward as shown in Figure 5. I also introduced a flag for each register in my code. They track if the register was written to and makes it possible to shorten a state file transformed from a witness to only the relevant registers. As they have no impact on the operation of the BTOR2 model, I will not mention them again.

4.2.3 Initialization

To initialize a state in BTOR2 from a RISC-V state file, the values in the registers must be loaded as constants, and for each memory address mentioned in the state file, the value and address has to be loaded as constants. Due to the inability to represent a full 64bit address space, the shrinking of the address space from state file to BTOR2 model must be handled. I decided to just initialize the addresses up to the BTOR2 model address space maximum and cut all others in the state file as I deem this the most predictable behavior. Everything not mentioned in the state file will be zero initialized. At last these constants must be used to initialize the state. For the registers this is straight forward, for the memory we must first write all memory addresses into a placeholder array which then we can use to initialize the real memory. Due to constraints in BTOR2, these constants have to be defined **before** the states, but initialization with the values must happen after the states. This means that this initialization process **wraps around** the state representation. The generation of constants is shown in Algorithm 2, whereas the actual initialization is shown in Algorithm 3.

4.2.4 Fetching the current instruction

To fetch the current instruction, I read the 4 bytes of the instruction and concatenate them as seen in Figure 6

4.2.5 Deconstruction of the instruction

Now having the instruction, we can deconstruct it to extract the *opcode*, rd, $rs1$, $rs2$, $funct3$, $funct7$ and imm. For everything apart from imm, this can be done by a shift and a masking. This is shown in Figure 7.

$truePc \leftarrow$ value of pc in state file
$maxPc \leftarrow$ number of addresses in BTOR2 model
pcValue $\leftarrow$ $truePc$ modulo $maxPc$
add to model:

n	constd	AS	*pcValue*	*pcConst*

for every register x_i **do**
  if register is initialised in state file **then**
    $registerValue \leftarrow$ value of x_i
    if $registerValue \neq 0$ **then**
      add to model:

n	constd	D	*registerValue*	x_i*Const*

    end
  end
end

add to model:

(n + 0)	state	Mem		*memPH*
(n + 1)	init	Mem	*memPH* (n + 0)	

$lastPH \leftarrow memPH$
$allInitialCells \leftarrow$ all initialised memory cells in the state file
$cutInitialCells \leftarrow$ remove all cells with address over maxPc
for every cell c in $cutInitialCells$ **do**
  $address \leftarrow$ address of c
  $value \leftarrow$ value of c
  add to model:

(n + 0)	constd	AS	*address*			
(n + 1)	constd	B	*value*			
(n + 2)	write	Mem	*lastPH*	(n + 0)	(n + 1)	*PHAfterC*

  $lastPH \leftarrow PHAfterC$
end
keep $lastPH$ for initialisation

Algorithm 2: Generating initialisation constants from state file in BTOR2

add to model:

n	init	AS	*pc*	*pcConst*

for every register x_i **do**
  if x_iConst was defined **then**
    add to model:

n	init	D	x_i	x_i*Const*

  end
end

add to model:

n	init	Mem	*memory*	*lastPh*

Algorithm 3: Initialising states in the BTOR2 model

(n + 0)	read	B	*memory*	*pc*	*instrB1*
(n + 1)	add	AS	*addressInc*	*pc*	*pc+1*
(n + 2)	read	B	*memory*	*pc+1*	*instrB2*
(n + 3)	add	AS	*addressInc*	*pc+1*	*pc+2*
(n + 4)	read	B	*memory*	*pc+2*	*instrB3*
(n + 5)	add	AS	*addressInc*	*pc+2*	*pc+3*
(n + 6)	read	B	*memory*	*pc+3*	*instrB4*
(n + 7)	concat	H	*instrB2*	*instrB1*	*instrH1*
(n + 8)	concat	H	*instrB4*	*instrB3*	*instrH2*
(n + 9)	concat	W	*instrH2*	*instrH1*	*instr*

Figure 6: Fetching the current instruction from memory

The immediate on the other hand must be first constructed from its subfields, which can be referenced in Figure 1. In the BTOR2 model this looks like in Figure 8. **(TODO: Reference to same method in riscvsim)** There are three things I want to point out:

First, some immediate subfields overlap exactly. I made use of this fact in lines (n + 1) with the overlap of $imm[11:5]$ of I- and S-type, and (n + 21) with J- and B-types $imm[10:5]$ overlap. Second, as described in Section 2.2 the immediate is always sign-extended. To archive this we make arithmetic right shifts, which do sign extension for us and with this pull our highest immediate bit to its correct place. Third, at line (n + 8), for sign extension we must shift right by 19. As this matches the opcode for arithmetic instructions with immediate, I used this and did not create a new constant.

Now I have *iTypeImm*, *sTypeImm*, *bTypeImm*, *uTypeImm* and *jTypeImm*. But it would be easier to just have one node *imm* where we can reference the immediate value regardless of the instruction. This is done in Figure 9, where first I defined booleans which check all opcodes that are neither R-type nor I-type. Then I chained if-then-else nodes to catch instructions that are of J-type, U-Type, B-Type or S-type. If the instruction is none of them, I can safely default to I-type as R-type does not handle with an immediate value. At the end I extend *imm* to the 64bit RV64I demands.

At this point I can also extract the values of the designated $rs1$ and $rs2$ registers. I show this for $rs1$ in Figure 4, it is the same for $rs2$ except that the names must be changed to $rs2$. Also, the comparison constants can be left out as they are already defined for $rs1$ and can be referenced from there.

(n + 0)	and	W	*instr*	*7Bitmask*	*opcode*
(n + 1)	srl	W	*instr*	*shiftToRd*	*rdPre*
(n + 2)	and	W	*rdPre*	*5Bitmask*	*rd*
(n + 3)	srl	W	*instr*	*shiftToRs1*	*rs1Pre*
(n + 4)	and	W	*rs1Pre*	*5Bitmask*	*rs1*
(n + 5)	srl	W	*instr*	*shiftToRs2*	*rs2Pre*
(n + 6)	and	W	*rs2Pre*	*5Bitmask*	*rs2*
(n + 7)	srl	W	*instr*	*shiftToFunct3*	*funct3Pre*
(n + 8)	and	W	*funct3Pre*	*shiftRd*	*funct3*
(n + 9)	srl	W	*instr*	*shiftToFunct7*	*funct7*

Figure 7: Extraction of values from the instruction without *imm*

for i from 1 to 31 **do**
  add to model:

n	eq	Bool	*rs1*	*iConst*	*isRs1Xi*

end
add to model:

n	ite	D	*isRs1X1*	*x1*	*x0*	*checkX1*

for i from 2 to 30 **do**
  add to model:

n	ite	D	*isRs1Xi*	*xi*	*checkX(i - 1)*	*checkXi*

end
add to model:

n	ite	D	*isRs1X31*	*x31*	*checkX30*	*rs1val*

Algorithm 4: Extracting the value of the register designated by $rs1$

(n + 0)	sra	W	*instr*	*shiftToRs2*	*iTypeImm*
(n + 1)	and	W	*iTypeImm*	*-5Bitmask*	*s[11:5]*
(n + 2)	add	W	*s[11:5]*	*rd*	*sTypeImm*
(n + 3)	and	W	*rd*	*-bitPicker*	*b[4:0]*
(n + 4)	and	W	*funct7*	*6Bitmask*	*b[10:5]Pre*
(n + 5)	sll	W	*b10:5Pre*	*shiftBy5*	*b[10:5]*
(n + 6)	and	W	*bitPicker*	*rd*	*b[11]Pre*
(n + 7)	sll	W	*b[11]Pre*	*shiftBy11*	*b[11]*
(n + 8)	sra	W	*instr*	*mathI*	*b[31:12]Pre*
(n + 9)	and	W	*b[31:12]Pre*	*12Bitmask*	*b[31:12]*
(n + 10)	add	W	*b[10:5]*	*b[4:0]*	*b[10:0]*
(n + 11)	add	W	*b[11]*	*b[10:0]*	*b[11:0]*
(n + 12)	add	W	*b[31:12]*	*b[11:0]*	*bTypeImm*
(n + 13)	and	W	*instr*	*-12Bitmask*	*uTypeImm*
(n + 14)	and	W	*rs2*	*-bitPicker*	*j[4:0]*
(n + 15)	and	W	*rs2*	*bitPicker*	*j[11]Pre*
(n + 16)	sll	W	*j[11]Pre*	*shiftBy11*	*j[11]*
(n + 17)	sll	W	*funct3*	*shiftToFunct3*	*j[14:12]*
(n + 18)	sll	W	*rs1*	*shiftToRs1*	*j[19:15]*
(n + 19)	sra	W	*instr*	*shiftBy11*	*j[31:20]Pre*
(n + 20)	and	W	*j[31:20]Pre*	*-20Bitmask*	*j[31:20]*
(n + 21)	add	W	*b[10:5]*	*j[4:0]*	*j[10:0]*
(n + 22)	add	W	*j[11]*	*j[10:0]*	*j[11:0]*
(n + 23)	add	W	*j[14:12]*	*j[11:0]*	*j[14:0]*
(n + 24)	add	W	*j[19:15]*	*j[14:0]*	*j[19:0]*
(n + 25)	add	W	*j[31:20]*	*j[19:0]*	*jTypeImm*

Figure 8: Extraction of all *imm* types from the instruction

(n + 0)	eq	Bool	*opcode*	*store*		*isSType*
(n + 1)	eq	Bool	*opcode*	*branch*		*isBType*
(n + 2)	eq	Bool	*opcode*	*auipc*		*uType1*
(n + 3)	eq	Bool	*opcode*	*lui*		*uType2*
(n + 4)	or	Bool	*uType1*	*uType2*		*isUType*
(n + 5)	eq	Bool	*opcode*	*jal*		*isJType*
(n + 6)	ite	W	*isSType*	*sTypeImm*	*iTypeImm*	*checkS*
(n + 7)	ite	W	*isBType*	*bTypeImm*	*checkS*	*checkB*
(n + 8)	ite	W	*isUType*	*uTypeImm*	*checkB*	*checkU*
(n + 9)	ite	W	*isJType*	*jTypeImm*	*checkU*	*imm32*
(n + 10)	sext	D	*imm32*	*32*		*imm*

Figure 9: Finding the correct immediate by opcode

(*isJALR* already exists)					
n	and	Bool	*isLoad*	*is5Funct3*	*isLHU*
(n + 0)	consth	W	*20*		*SUBWf7*
(n + 1)	eq	Bool	*funct7*	*SUBWf7*	*fitsF7SUBW*
(n + 2)	and	Bool	*is0Funct3*	*fitsF7SUBW*	*fitsF3SUBW*
(n + 3)	and	Bool	*isLoad*	*fitsF3SUBW*	*isSUBW*

(TODO: Use subfigs)

Figure 10: Instruction detection of `JALR`, `LHU` and `SUBW` as described in Algorithm 5

4.2.6 Instruction Detection

For the next-state logic, the only thing left that we need to know is the actual current command. So I defined a check *isInstruction* for each instruction. As this is quite repetitive, Algorithm 5 describes a generalized approach to reach these booleans. An example for each instruction subgroup in Algorithm 5 can be found in Figure 10. Of course the funct7 checks from the $needsf7$ subgroup can be reused if multiple instructions use the same funct7.

add to model:

(n + 0)	eq	Bool	*opcode*	*load*	*isLoad*
(n + 1)	eq	Bool	*opcode*	*opImm*	*isOpImm*
(n + 2)	eq	Bool	*opcode*	*auipc*	*isAUIPC*
(n + 3)	eq	Bool	*opcode*	*opImm32*	*isOpImm32*
(n + 4)	eq	Bool	*opcode*	*store*	*isStore*
(n + 5)	eq	Bool	*opcode*	*op*	*isOp*
(n + 6)	eq	Bool	*opcode*	*lui*	*isLUI*
(n + 7)	eq	Bool	*opcode*	*op32*	*isOp32*
(n + 8)	eq	Bool	*opcode*	*branch*	*isBranch*
(n + 9)	eq	Bool	*opcode*	*jalr*	*isJALR*
(n + 10)	eq	Bool	*opcode*	*jal*	*isJAL*

for i from 0 to 7 **do**
  add to model:

n	eq	Bool	*funct3*	*iConst*	*isiFunct3*

end
onlyOp ← [LUI, AUIPC, JAL, JALR]
needsf7 ← [SRL, SRA, SRLI, SRAI, SRLW, SRAW, SRLWI, SRAWI, ADD, SUB, ADDW, SUBW]
rest ←[all other instructions]

for all instructions I in *onlyOp* **do**
  isI is already defined
end
for all instructions I in *rest* **do**
  opname ← opcode name of I
  f3val ← expected funct3 of I as digit
  add to model:

n	and	Bool	*isopname*	*isf3valFunct3*	*isI*

end
for all instructions I in *needsf7* **do**
  opname ← opcode name of I
  f3val ← expected funct3 of I as digit
  f7hex ← expected funct7 of I as hexadecimal number
  add to model:

(n + 0)	consth	W	*f7hex*		*If7*
(n + 1)	eq	Bool	*funct7*	*If7*	*fitsF7I*
(n + 2)	and	Bool	*isf3valFunct3*	*fitsF7I*	*fitsF3I*
(n + 3)	and	Bool	*isopname*	*fitsF3I*	*isI*

end

Algorithm 5: Generalised approach to instruction detection

n	add	D	*imm*	*pc*	*rdAUIPC*

11.1: AUIPC

Figure 11: Instruction execution for chosen instructions

4.2.7 Next-State Logic

The next state logic is basically the core of the model. Almost everything else works towards this point. The Goal is to create the changes each instruction would make and then only inserting the changes specific to the instruction in the state. Each state node in the model must have an accompanying next node to work as intended. But first the changed values are needed.

Creating all Values of Instruction execution

It would be too long and unnecessary to go through all instructions, as this is simply following the RV64I ISA, but I want to give an example for each group of instructions as they were divided in Table 1. I show this for `AUIPC`, `JALR`, `BEQ`, `LHU`, `SD`, `ANDI`, `SLLIW`, `SLT` and `SUBW` in Figure 11. In this examples one can see multiple overlaps which can be used, e.g. the addresses for load and store instructions or the 32bit versions of the word instructions. Also, I took `SD` to show that all other store instructions happen as interim results of preparing `SD`. It is similar with load instructions, but here we only get overlapping pre-results which each have to be sign extended to the expected 64bit on their own.

With this done we can sort each change to its instruction.

The next Memory

Defining the next memory array is simple. I just cascade through all store instructions with if-then-else nodes and by setting the final 'else' as the current memory array, if

(n + 0)	add	AS	*pc*	*pcInc*	*nextPc*
(n + 1)	add	D	*imm*	*rs1val*	*pcJALR64pre*
(n + 2)	and	D	*-1Const*	*pcJALR64pre*	*pcJALR64*
(n + 3)	slice	AS	*pcJALR64*	*15*	*pcJALR*
(n + 4)	uext	D	*nextPc*	*48*	*rdJALR*

(TODO: pc overflow erwähnen)

11.2: JALR

Figure 11: Instruction execution for chosen instructions

(n + 0)	add	AS	*pc*	*pcInc*		*nextPc*
(n + 1)	slice	AS	*imm*	*15*	*0*	*ImmAS*
(n + 2)	add	AS	*pc*	*ImmAS*		*pcBranch*
(n + 3)	eq	Bool	*rs1val*	*rs2val*		*isBEQcond*
(n + 4)	ite	AS	*isBEQcond*	*pcBranch*	*nextPc*	*pcBEQ*

11.3: BEQ

Figure 11: Instruction execution for chosen instructions

(n + 0)	add	D	*rs1val*	*imm*		*1stAddrPre*
(n + 1)	slice	AS	*1stAddrPre*	*15*	*0*	*1stAddr*
(n + 2)	add	AS	*1stAddr*	*addressInc*		*2ndAddr*
(n + 3)	read	B	*memory*	*1stAddr*		*loadB1*
(n + 4)	read	B	*memory*	*2ndAddr*		*loadB2*
(n + 5)	concat	H	*loadB2*	*loadB1*		*loadB2B1*
(n + 6)	uext	D	*loadB2B1*	*48*	*0*	*rdLHU*

11.4: LHU

Figure 11: Instruction execution for chosen instructions

(n + 0)	add	D	*rs1val*	*imm*		*1stAddrPre*
(n + 1)	slice	AS	*1stAddrPre*	*15*	*0*	*1stAddr*
(n + 2)	add	AS	*1stAddr*	*addressInc*		*2ndAddr*
(n + 3)	add	AS	*2ndAddr*	*addressInc*		*3rdAddr*
(n + 4)	add	AS	*3rdAddr*	*addressInc*		*4thAddr*
(n + 5)	add	AS	*4thAddr*	*addressInc*		*5thAddr*
(n + 6)	add	AS	*5thAddr*	*addressInc*		*6thAddr*
(n + 7)	add	AS	*6thAddr*	*addressInc*		*7thAddr*
(n + 8)	add	AS	*7thAddr*	*addressInc*		*8thAddr*
(n + 9)	slice	B	*rs2val*	*7*	*0*	*storeB1*
(n + 10)	slice	B	*rs2val*	*15*	*8*	*storeB2*
(n + 11)	slice	B	*rs2val*	*23*	*16*	*storeB3*
(n + 12)	slice	B	*rs2val*	*31*	*24*	*storeB4*
(n + 13)	slice	B	*rs2val*	*39*	*32*	*storeB5*
(n + 14)	slice	B	*rs2val*	*47*	*40*	*storeB6*
(n + 15)	slice	B	*rs2val*	*55*	*48*	*storeB7*
(n + 16)	slice	B	*rs2val*	*63*	*56*	*storeB8*
(n + 17)	write	Mem	*memory*	*1stAddr*	*storeB1*	*memorySB*
(n + 18)	write	Mem	*memorySB*	*2ndAddr*	*storeB2*	*memorySH*
(n + 19)	write	Mem	*memorySH*	*3rdAddr*	*storeB3*	*memoryB3*
(n + 20)	write	Mem	*memoryB3*	*4thAddr*	*storeB4*	*memorySW*
(n + 21)	write	Mem	*memorySW*	*5thAddr*	*storeB5*	*memoryB5*
(n + 22)	write	Mem	*memoryB5*	*6thAddr*	*storeB6*	*memoryB6*
(n + 23)	write	Mem	*memoryB6*	*7thAddr*	*storeB7*	*memoryB7*
(n + 24)	write	Mem	*memoryB7*	*8thAddr*	*storeB8*	*memorySD*

11.5: SD

Figure 11: Instruction execution for chosen instructions

n	and	D	*rs1val*	*imm*	*rdANDI*

11.6: ANDI

Figure 11: Instruction execution for chosen instructions

(n + 0)	and	W	*imm32*	*5Bitmask*		*shamtIW*
(n + 1)	slice	W	*rs1val*	*31*	*0*	*rs1val32*
(n + 2)	sll	W	*rs1val32*	*shamtIW*		*rdSLLIWpre*
(n + 3)	sext	D	*rs1val32*	*32*		*rdSLLIW*

11.7: SLLIW

Figure 11: Instruction execution for chosen instructions

(n + 0)	slt	Bool	*rs1val*	*rs2val*	*rdSLTpre*
(n + 1)	uext	D	*rdSLTpre*	*63*	*rdSLT*

11.8: SLT

Figure 11: Instruction execution for chosen instructions

(n + 0)	slice	W	*rs1val*	*31*	*0*	*rs1val32*
(n + 1)	slice	W	*rs2val*	*31*	*0*	*rs2val32*
(n + 2)	sub	W	*rs1val32*	*rs2val32*		*rdSUBWpre*
(n + 3)	sext	D	*rdSUBWpre*	*32*		*rdSUBW*

11.9: SUBW

Figure 11: Instruction execution for chosen instructions

no 'if' catches, the array is not changed. All this is shown in Figure 12.

The next pc

For the next pc it looks mostly the same as shown in Figure 13. Only the behavior if no 'if' catches is different as pc must point to the next instruction to execute. This nextPc was already computed for the JAL and JALR instructions, so I reused it. The unconditional jumps also change the value in rd, but this is done in the next subsection.

The next rd

At last the x registers must be updated. The procedure is defined in Figure 6. With exception to x0 this is the same for all these registers. Also, it is similar in its

(n + 0)	ite	Mem	*isSB*	*memorySB*	*memory*	*newMem3*
(n + 1)	ite	Mem	*isSH*	*memorySH*	*newMem3*	*newMem2*
(n + 2)	ite	Mem	*isSW*	*memorySW*	*newMem2*	*newMem1*
(n + 3)	ite	Mem	*isSD*	*memorySD*	*newMem1*	*newMem*
(n + 4)	next	Mem	*memory*	*newMem*		

Figure 12: Next-State logic for the memory array

(n + 0)	ite	AS	*isBGEU*	*pcBGEU*	*nextPc*	*newPc7*
(n + 1)	ite	AS	*isBLTU*	*pcBLTU*	*newPc7*	*newPc6*
(n + 2)	ite	AS	*isBGE*	*pcBGE*	*newPc6*	*newPc5*
(n + 3)	ite	AS	*isBLT*	*pcBLT*	*newPc5*	*newPc4*
(n + 4)	ite	AS	*isBNE*	*pcBNE*	*newPc4*	*newPc3*
(n + 5)	ite	AS	*isBEQ*	*pcBEQ*	*newPc3*	*newPc2*
(n + 6)	ite	AS	*isJALR*	*pcJALR*	*newPc2*	*newPc1*
(n + 7)	ite	AS	*isJAL*	*pcJAL*	*newPc1*	*newPc*
(n + 8)	next	AS	*pc*	*newPc*		

Figure 13: Next-State logic for the pc register

procedure as defining the next memory or pc but instead of a handful of instructions, I have to go over 39 of them as only branch and store instructions do not change rd. Because of this, I took the liberty to not exactly show the cascade for all relevant instructions in Algorithm 6 but only indicate it.

add to model:

n	next	D	*x0*	*x0*

for i from 1 to 31 **do**

add to model:

(n + 0)	ite	D	*isLUI*	*rdLUI*	*xi*	*newXi-49*
⋮	ite	D	⋮	⋮	⋮	⋮
(n + 47)	ite	D	*isSRAW*	*rdSRAW*	*newXi-2*	*newXi-1*
(n + 48)	eq	Bool	*rd*	*iConst*		*isRdXi*
(n + 49)	ite	D	*isRdXi*	*newXi-1*	*xi*	*newXi*
(n + 50)	next	D	*xi*	*newXi*		

end

Algorithm 6: Next-state logic for all x registers

4.2.8 Constraints

The only thing left is to define constraints to end the model checker. First is the intended end of reaching a set number of Iterations. It is shown in Figure 14.

After this I defined some extra constraints to check for bad instructions. First is

```
(n + 0)  one     D                                  counterInc
(n + 1)  constd  D     nIterations                  maxIterations
(n + 2)  state   D                                  counter
(n + 3)  init    D     counter     emptyReg
(n + 4)  add     D     counter     counterInc       newCounter
(n + 5)  next    D     counter     newCounter
(n + 6)  eq      Bool  counter     maxIterations    isMaxIter
(n + 7)  bad           isMaxIter
```

Figure 14: Constraining the model by iteration count

checked if the opcode is valid for my model. The second constraint catches if the instruction can not be detected even whilst the opcode is valid. This is shown in Figure 15. The constraint in Figure 16 handles instruction-address-misaligned exceptions for jump instructions.

Of course other constraints can be defined. Options would be to stop on a specific pc or if a register reaches a specified value.

(TODO: Maybe add examples on how to do?)

(TODO: Maybe internal references in figures should be numbers...)

4.3 Testing for Correctness

To test my model, I compared its results to my RISC-V simulators (Section 2.3) results.

With a given state, both the simulation and the BTOR2 model are run. For both the iteration maximum is set to 1. The resulting BTOR2 witness can not be directly compared to the resulting state of the simulation. So I also implemented a simple converter from witness to state [7, src/restate_witness.c]. These two states now can be compared. I have written a shell script for this at [7, sh_utils/compare_iterations.sh]

(n + 0)	or	Bool	*isLoad*	*isOpImm*	*isOpcodeValid9*
(n + 1)	or	Bool	*isAUIPC*	*isOpcodeValid9*	*isOpcodeValid8*
(n + 2)	or	Bool	*isOpImm32*	*isOpcodeValid8*	*isOpcodeValid7*
(n + 3)	or	Bool	*isStore*	*isOpcodeValid7*	*isOpcodeValid6*
(n + 4)	or	Bool	*isOp*	*isOpcodeValid6*	*isOpcodeValid5*
(n + 5)	or	Bool	*isLUI*	*isOpcodeValid5*	*isOpcodeValid4*
(n + 6)	or	Bool	*isOp32*	*isOpcodeValid4*	*isOpcodeValid3*
(n + 7)	or	Bool	*isBranch*	*isOpcodeValid3*	*isOpcodeValid2*
(n + 8)	or	Bool	*isJALR*	*isOpcodeValid2*	*isOpcodeValid1*
(n + 9)	or	Bool	*isJAL*	*isOpcodeValid1*	*isOpcodeValid*
(n + 10)	bad		-*isOpcodeValid*		
(n + 11)	or	Bool	*isLUI*	*isAUIPC*	*isInstrValid47*
(n + 12)	or	Bool	*isJAL*	*isInstrValid47*	*isInstrValid46*
⋮	or	Bool	⋮	⋮	⋮
(n + 58)	or	Bool	*isSRAW*	*isInstrValid1*	*isInstrValid*
(n + 59)	and	Bool	-*isInstrValid*	*isOpcodeValid*	*unknownInstr*
(n + 60)	bad		*unknownInstr*		

Figure 15: Constraining the model on unknown instructions

(n + 0)	zero	AS			*pcZero*
(n + 1)	constd	AS	*3*		*pcBitmask*
(n + 2)	and	AS	*pcBitmask*	*pcJAL*	*lowbitsJAL*
(n + 3)	and	AS	*pcBitmask*	*pcJALR*	*lowbitsJALR*
(n + 4)	and	AS	*pcBitmask*	*pcBEQ*	*lowbitsBEQ*
⋮	and	AS	*pcBitmask*	⋮	⋮
(n + 9)	and	AS	*pcBitmask*	*pcBGEU*	*lowbitsBGEU*
(n + 10)	neq	Bool	*pcZero*	*lowbitsJAL*	*pcMsaJAL*
(n + 11)	neq	Bool	*pcZero*	*lowbitsJALR*	*pcMsaJALR*
(n + 12)	neq	Bool	*pcZero*	*lowbitsBEQ*	*pcMsaBEQ*
⋮	neq	Bool	*pcZero*	⋮	⋮
(n + 17)	neq	Bool	*pcZero*	*lowbitsBGEU*	*pcMsaBGEU*
(n + 18)	or	Bool	*pcMsaJAL*	*pcMsaJALR*	*pcMsa6*
(n + 19)	or	Bool	*pcMsaBEQ*	*pcBEQ*	*pcMsa5*
⋮	or	Bool	⋮	⋮	⋮
(n + 24)	or	Bool	*pcMsaBGEU*	*pcMsa1*	*pcMsa*
(n + 25)	bad		*pcMsa*		

Figure 16: Constraining the model on misaligned addresses

To generate RISC-V states, I implemented a fuzzer [7, src/state_fuzzer.c] to generate randomized states with one valid instruction at the address of pc. The fuzzer first chooses an instruction to test, on this basis it fills all variable parts of the instruction, e.g. rd or imm. Now all registers relevant to the instruction are filled with a random 64bit value. Also, a pc value is generated so that the instruction still fits in the limited address space of the BTOR2 model. At last if a jump instruction was chosen, a possible address misalignment is fixed and an address overflow prevented. The second part is for simplifying later comparison on the resulting states, as now a correct execution of the instruction always results in the exact same resulting state although the differences between simulation and the BTOR2 model.

With this it is possible to start test series. For this I implemented a shell script, too [7, sh_utils/test_btor2_model.sh]. Also, as with big amounts of tests, it becomes harder to keep an overview over failed tests. To counter this I also have written a script to unite all failed tests into one file and also add some not so easy to access information like instruction name or immediate value [7, sh_utils/diff_logger.sh].

I have run around 5,000,000 tests on this model without one failing, so I assume that my implementation is correct.

5 Benchmarks

With a model implemented, I can test how good it runs. I run my benchmarks on an Intel Core i5-6200U with the btormc model checker published with the BTOR2 format [2]. Each test was run five times and the resulting times averaged. I also tried to run the tests with the model checkers avr and pono **(TODO: cite)** as they placed first and second in the hardware model checking competition of 2024, but I will talk about them at the results.

5.1 Tests

I devised two base tests formed from four RISC-V instructions as shown in Figure 17. I chose to implement one test with and one without memory operation to be able to measure the impact memory operations have in the model. For this reason the program for these tests are very similar. Both have three instructions forming a loop and one instruction as a "workhorse". My naming for these also derive from this instruction as add and write_mem. This program is set into a state. An example for this is found in Figure 18, where the add program is set up for 256 loops. In this, x1 acts as a loop limiter, x2 as a loop counter and x3 as an accumulator. The instructions are set in the first bytes of the memory. In difference to this, the memory operation test uses x2 also as an address to store the first byte of a register and x3 as this register. Both tests are also implemented with rising loop counts up to 2048 to form a reference on how they behave with more iterations to process.

```
        bge x2 x1 0x10          jump out of program if x1 = x2
add x3 x3 x2 |                  either (add counter onto x3)
             | sb x3 0x14(x2)   or (store the first byte of x3 at counter + 0x14)
        addi x2 x2 0x1          increment counter in x2
        jalr x0 x0 0x0          jump back to address 0
```

Figure 17: Base test cases for the benchmarks

```
REGISTERS:
PC:0
x1:100
x2:0

MEMORY:
0:001158E3 # BGE x2 x1 0x10
4:002181B3 # ADD x3 x3 x2
8:00110113 # ADDI x2 x2 1
c:00000067 # JALR x0 x0 0
```

Figure 18: Example state for benchmark add_0256

Further on I tested the impact of initialized memory on the runtime of the test. For this I added tests with the prefix fullmem, where I filled the memory addresses 0x18 to 0xfff with the pattern '0101'. The first four words of the memory are not filled for there lies the program of the test. The fifth could also be filled instead of being left zero-initialized, but I decided against it because the test is designed to terminate by jumping to this address and not finding a valid instruction there. Leaving it zero-initialized guarantees this. To further extend on this test, I also added some tests on double the initialized memory, so filled to the address 0x1fff.**(TODO: benchmark them, add times)**

I also tested on the impact of the address space size. For this I generated models from add_0256, add_1024, write_mem_0256 and write_mem_1024 with extended address space. These tests have the prefix "extaddr_x", where x is the maximum length of an address in this model in bits.

Finally, I also implemented the base add tests similar to the BTOR2 model that

F.Schrögendorfer describes in his master thesis [3, Chapter 8]. These tests have the prefix “nopc”, as the pc is abstracted to activation flags for each instruction. I show the generation of the model on the example of nopc_add_0256. First of, we need the sorts necessary for RISC-V and the constants necessary for the model:

```
 1  sort    bitvec  1     bool
 2  sort    bitvec  8     memcell
 3  sort    bitvec  16    addressspace
 4  sort    bitvec  64    register
 5  sort    array   3 2   Memory
 6  zero    1             false
 7  one     1             true
 8  zero    2             emptymem
 9  zero    4             emptyreg
10  zero    3             pcinit
11  consth  3       4     pcinc
12  consth  3       10    instr1_pcmod
13  consth  4       100   nloops
```

With this now the registers and memory can be defined:

```
 99  state  3  pc      110  state  4  x10     121  state  4  x21
100  state  4  x0      111  state  4  x11     122  state  4  x22
101  state  4  x1      112  state  4  x12     123  state  4  x23
102  state  4  x2      113  state  4  x13     124  state  4  x24
103  state  4  x3      114  state  4  x14     125  state  4  x25
104  state  4  x4      115  state  4  x15     126  state  4  x26
105  state  4  x5      116  state  4  x16     127  state  4  x27
106  state  4  x6      117  state  4  x17     128  state  4  x28
107  state  4  x7      118  state  4  x18     129  state  4  x29
108  state  4  x8      119  state  4  x19     130  state  4  x30
109  state  4  x09     120  state  4  x20     131  state  4  x31
```

Note that I skipped the node IDs of 14-98. As BTOR2 needs unique but not continouus node IDs, I took this liberty so each register node ID equals the register nuber plus 100. By this it becomes easier to write a model by hand.
Next the initial memory is placed in the model:

```
144  state   5
145  init    5  14 8
146  consth  3  0
147  consth  2  e3
148  write   5  144 146 147
149  consth  3  1
150  consth  2  58
151  write   5  148 149 150
152  consth  3  2
153  consth  2  11
154  write   5  151 152 153
155  consth  3  3
156  consth  2  00
157  write   5  154 155 156
158  consth  3  4
159  consth  2  b3
160  write   5  157 158 159
161  consth  3  5
162  consth  2  81
163  write   5  16 161 162
164  consth  3  6
165  consth  2  21
166  write   5  16 164 165
167  consth  3  7
168  consth  2  00
169  write   5  166 167 168
170  consth  3  8
171  consth  2  13
172  write   5  169 170 171
173  consth  3  9
174  consth  2  01
175  write   5  172 173 174
176  consth  3  a
177  consth  2  11
178  write   5  175 176 177
179  consth  3  b
180  consth  2  00
181  write   5  178 179 180
182  consth  3  c
183  consth  2  67
184  write   5  181 182 183
185  consth  3  d
186  consth  2  00
187  write   5  184 185 186
188  consth  3  e
189  consth  2  00
190  write   5  187 188 189
191  consth  3  f
192  consth  2  00
193  write   5  190 191 192
```

Due to constraints in BTOR2, the initial memory must be placed before the intended memory state node. Node 144 is again a state only for initializing the memory, as I previously did in the initialisation for my own model found in Section 4.2.3.
Now the real memory state can be defined. With this I also define the flags for each instruction. I changed their name from stmt (Schrögendorfer) to instr for increased clarity. An exit code is not necessary as I do not need differentiation between terminations of the model, more so I am only interested *when* the model terminates.

```
199  state  5  memory
200  state  1  instr0
201  state  1  instr1
202  state  1  instr2
203  state  1  instr3
204  state  1  endflag
```

Next up would be the inputs and constraints for these. Both are not needed for the RISC-V model because neither are threads needed as I do not model parallel processing, nor is flushing as I use a naive memory model. And without inputs, the

constraints are also nonexistent. So really, next up is initialization and transitions of the states. First is pc:

```
300  init  3  99   10          | pc is zero initialized
301  add   3  99   11          | normal pc operation
302  add   3  99   12          | instr1 jump
303  eq    1  101  102         | instr1 branch condition
304  ite   3  303  302  301    | IF condition THEN jump ELSE increment
305  ite   3  200  304  99     | IF instr1 THEN check ELSE leave pc
306  or    1  201  202         | instr2-3
307  ite   3  306  301  305    | IF instr2-3 THEN increment ELSE instr1
308  ite   3  203  10   307    | IF instr4 THEN jump0 ELSE try instr2-3
309  next  3  99   308
```

As the transition for pc are quite possibly the most complex I added some explanation. x0 and x1 are skipped for now, so next registers x2 and x3:

```
314  init  4  102  9
315  one   4
316  add   4  102  315
317  ite   4  202  316  102
318  next  4  102  317

319  init  4  103  9
320  add   4  102  103
321  ite   4  201  320  103
322  next  4  103  321
```

x2 increments by one each time the third instruction is run and x3 adds x2 every time the second instruction is run. All other registers and the memory do not change during execution, so I show them in one big block:

```
310  init 4 100 9
311  next 4 100 100
312  init 4 101 13
313  next 4 101 101
323  init 4 104 9
324  next 4 104 104
325  init 4 105 9
326  next 4 105 105
327  init 4 106 9
328  next 4 106 106
329  init 4 107 9
330  next 4 107 107
331  init 4 108 9
332  next 4 108 108
333  init 4 109 9
334  next 4 109 109
335  init 4 110 9
336  next 4 110 110
337  init 4 111 9
338  next 4 111 111
339  init 4 112 9
340  next 4 112 112
341  init 4 113 9
342  next 4 113 113
343  init 4 114 9
344  next 4 114 114
345  init 4 115 9
346  next 4 115 115
347  init 4 116 9
348  next 4 116 116
349  init 4 117 9
350  next 4 117 117
351  init 4 118 9
352  next 4 118 118
353  init 4 119 9
354  next 4 119 119
355  init 4 120 9
356  next 4 120 120
357  init 4 121 9
358  next 4 121 121
359  init 4 122 9
360  next 4 122 122
361  init 4 123 9
362  next 4 123 123
363  init 4 124 9
364  next 4 124 124
365  init 4 125 9
366  next 4 125 125
367  init 4 126 9
368  next 4 126 126
369  init 4 127 9
370  next 4 127 127
371  init 4 128 9
372  next 4 128 128
373  init 4 129 9
374  next 4 129 129
375  init 4 130 9
376  next 4 130 130
377  init 4 131 9
378  next 4 131 131
379  init 5 199 193
380  next 5 199 199
```

Now only the instruction flags are left to handle:

```
381  init  1    200 7      | instr0 executes initally
382  next  1    200 203    | instr0 only after instr3
383  init  1    201 6      |
384  and   1    200 -303   | instr0 and no branch
385  next  1    201 384    | instr1 only after non-branching instr0
386  init  1    202 6      |
387  next  1    202 201    | instr2 only after instr2
388  init  1    203 6      |
389  next  1    203 202    | instr3 only after instr3
390  init  1    204 6      |
391  and   1    200 303    |
392  next  1    204 391    | endflag if instr0 branches
400  bad   204             | endflag terminates
```

And with this, the model is can be run. The whole suite of add tests can be derived from this model by changing the constant value of node 13 to the appropriate loop count.

5.2 Results

To start I placed all iterations based benchmarks with btormc into Table 3 and the benchmarks of extended address space into Table 4. From Table 4 one can assume that the extension of address space does not impact the runtime of the model checker in a meaningful way. As seen in Table 5, with rising loop counts, memory operations need increasingly more time in comparison to their respective non-memory operation benchmarks. Also with rising loop counts the impact of large amounts of memory initialised is reduced, which was to be expected.

The nopc benchmark is significantly faster than my model. This was to be expected, as Schrögendorfer models one RISC-V program specifically whereas I model the processor and feed it different programs by initialization.

loops	base		fullmem		nopc
	add	writemem	add	writemem	add
0256	2.635	2.877	9.759	13.344	0.136
0512	6.195	7.306	16.402	24.209	0.268
0768	10.802	13.283	24.093	36.388	0.414
1024	16.306	21.004	32.732	50.376	0.569
1280	23.032	30.200	42.410	65.746	0.728
1536	30.669	41.262	52.961	83.036	0.898
1792	39.463	53.940	64.598	101.475	1.075
2048	48.944	68.521	77.084	122.189	1.276

Table 3: Times of iterations based benchmarks

bits of address space	16	17	18	19	20
add_0256	2.635	2.632	2.626	2.626	2.624
add_1024	16.306	16.464	16.511	16.452	16.460
writemem_0256	2.877	2.88	2.890	2.889	2.890
writemem_1024	21.004	21.131	21.215	21.181	21.163

Table 4: Times of extended address space benchmarks

loops	$\frac{add}{writemem}$	$\frac{fullmem\_add}{fullmem \quad writemem}$	$\frac{add}{fullmem \quad add}$	$\frac{writemem}{fullmem \quad writemem}$	$\frac{add}{nopc \quad add}$
0256	0.92	0.73	0.27	0.22	19.38
0512	0.85	0.68	0.38	0.3	23.12
0768	0.81	0.66	0.45	0.37	26.09
1024	0.78	0.65	0.5	0.42	28.66
1280	0.76	0.65	0.54	0.46	31.64
1536	0.74	0.64	0.58	0.5	34.15
1792	0.73	0.64	0.61	0.53	36.71
2048	0.71	0.63	0.63	0.56	38.36

Table 5: Relative runtime of benchmarks

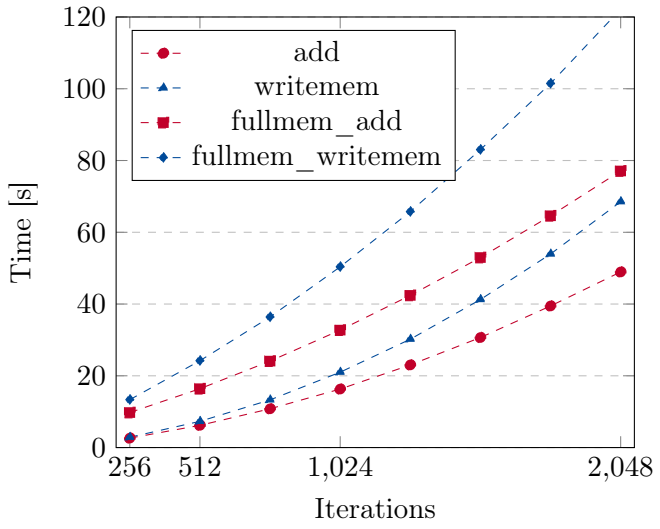

Figure 19: Table 3 plotted

6 Conclusion

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.

[6] "RISC-V-Simulator," https://github.com/Kr1mo/Risc-V-Simulator.

[7] "RISC-V_to_BTOR2," https://github.com/Kr1mo/RISC-V_to_BTOR2.

(TODO: Add repo versions)