# Metamorphic Testing for Processor Verification: A RISC-V Case Study at the Instruction Level

Frank Riese[1] Vladimir Herdt[1,2] Daniel Große[1,3] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[3]Institute for Complex Systems, Johannes Kepler University Linz, Austria
Frank.Riese@dfki.de, vherdt@uni-bremen.de, daniel.grosse@jku.at, drechsler@uni-bremen.de

***Abstract*—*Metamorphic Testing* (MT) has been shown to be a very effective technique in the *Software* (SW) domain. MT does not require a reference model to compare against for testing but instead relies on *Metamorphic Relations* (MR) to derive the expected result from relationships between several calls to the function under test. An example of an MR is the expectation that the sum of an arbitrary list of integers remain unchanged regardless of it being sorted or reversed. Thus, a key requirement for applying MT effectively is availability of MRs specific to the domain at hand.**

**In this paper, we propose MT to the domain of processor verification. As a case study, we consider the RISC-V *Instruction Set Architecture* (ISA) and provide MRs tailored for RISC-V. For evaluation purposes, we propose an efficient on-the-fly MT framework that integrates the MRs with an *Instruction Set Simulator* (ISS). We measure the quality of those MRs by the number of mutations they kill, also referred to as mutation analysis. Our experiments demonstrate the effectiveness of the MRs to kill all mutations, which confirms our research question that MT is also a suitable technique for the domain of processor verification.**

## I. Introduction

*Metamorphic Testing* (MT) is an approach to testing that has, on many occasions, been shown to have the capacity for revealing many faults other testing techniques had not. Those faults are not limited to simplistic or purely academic systems but have been making a growing industrial impact on flagship products of reputable companies, such as Google Maps and Bing [1]. Segura et al. cite many examples, including: "data engineering, simulation and modeling, compilers, machine learning programs, autonomous cars and drones, and cyber-security" [2]. In Le et al. a remarkable 147 confirmed bugs have been found by its authors, through the use of MT, for the GCC and LLVM suites of compilers, two pieces of software that are notable for the breadth of their use and significant in how fundamental their proper functioning is to the production of other software [3]. Zhou et al. cite a vivid example in which they had located and reported the bug in a system for self-driving cars responsible for the fatal accident involving a pedestrian, infamously world-wide the first such case – eight days before the tragic event [4]. Since 2016, there is now an international workshop dedicated to the topic of MT, Segura et al. have elevated MT to being referred to as a "fully-fledged testing paradigm" [2] and the number of publications covered in a notable survey paper roughly clock in at a respectable 150 publications [5].

The remarkable success of MT has motivated us to explore application of MT to the domain of processor verification, an area in that MT has found comparatively little application thus far. To this end, we test the *Instruction Set Architecture* (ISA) of a concrete, modern processor architecture through the use of MT and present the results of experimental evaluations that demonstrate the effectiveness of our approach. We have chosen RISC-V as our target, an open and freely available standard that is showing a rapid rate of adoption and great promise for future use.

Defining part of MT is the use of a *Metamorphic Relation* (MR), so called because they relate inputs and outputs of multiple executions of the same *System Under Test* (SUT) to each other to determine if a *Metamorphic Test Case* (MTC) has passed, as opposed to comparing a singular response from the SUT against a test oracle. By MTC we mean the concrete implementation of a test case that uses the abstract relation of an MR as part of its pass criteria. An example of such a relation is that the value of the sine function repeats every $2\pi$ so that we can assert $sin(x) = sin(x+2\pi)$, an MR, without having to know what the actual value, a test oracle, of either one is. This is a huge advantage over other test techniques and the literature frequently describes this as a way by which to avoid the "oracle problem" [6] [7]. One consequence of this, and further motivation for us, is that we can reduce the frequent reliance on reference models for processor verification, which represent an instance of such a test oracle.

Among our main contributions are a set of such MRs, tailored to the RISC-V ISA on the basis of its specifications [8] [9], and their implementation into MTCs. We cover the majority of instructions from the RV32I ISA, a mandatory set that constitutes the core of the RISC-V ISA.

In order to assess the quality of our MRs, we follow up with an experimental evaluation. For this we use mutation analysis, a powerful evaluation approach that finds frequent application in combination with MT [5]. This means that we inject point changes, so-called mutations, into an SUT to produce variations, so-called mutants, that deviate from expected behavior. The goal of a test set is to identify as many of those mutants as possible, the ratio of which is called the mutation score. These mutations are usually generated by picking replacements from sets called mutation classes. Here, too, we have found that the area of application has a bearing on which classes make sense, so we have customized our mutation classes to RISC-V.

Therefore, we introduce an MT framework that executes our test cases, efficiently injects hundreds of mutations into a RISC-V *Instruction Set Simulator* (ISS), and collects a variety of metrics about performance of our MTCs. We reach a perfect mutation score of 100%, meaning that of our roughly 400 mutations none survive. From this we conclude that our approach can be effectively used to apply MT to processor verification. All relevant source code has been made available under https://github.com/agra-uni-bremen/riscv-metamorphic-testing.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001 and within the project Scale4Edge under contract no. 16ME0127.
978-1-6654-2614-5/21/$31.00 ©2021 IEEE

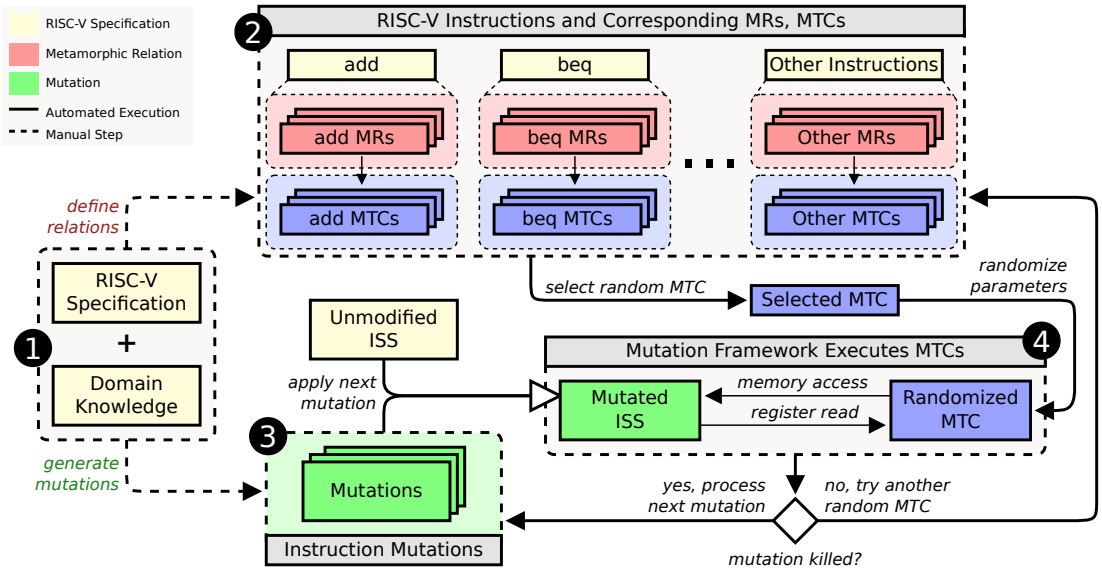


Figure 1: Overview of our proposed MT approach

## II. Related Work

We already discussed applications of MT in various software verification related domains as part of our introduction. In the hardware verification domain, on the other hand, the number of proposed MT approaches is much more limited thus far. We are only aware of [10] and [11], which leverage MT to verify the behavior of radio frequency amplifiers and test for hardware fault tolerance, respectively.

Regarding verification for RISC-V specifically, there exist a number of test generation approaches that leverage constraints [12], fuzzing [13] and other randomization techniques [14], as well as formal approaches such as the *OneSpin 360 DV* RISC-V verification app [15] and *riscv-formal* [16]. However, they require the availability of a reference model or property set which acts as a test oracle.

We are not aware of any approach that proposes MT for RISC-V or processor verification in general. We demonstrate for the first time that MT is a viable alternative in this domain.

## III. Metamorphic Testing with RISC-V Case Study

In this section we present our proposed MT approach tailored for RISC-V. We start with an overview.

### A. Overview

Figure 1 provides an overview of our approach. There, four major stages of the method by which we have proceeded are marked with ❶ to ❹. In ❶, we consult the RISC-V specification and use our knowledge of the hardware domain to derive relations and mutations. In ❷, MRs are defined on a per-instruction basis. In ❸, mutations are defined, to be then injected into the ISS. In ❹, our framework executes MTCs and keeps track of killed mutations. These mutations are injected into the ISS, one at a time. MTCs are selected one by one and executed with randomized parameters. Randomization is used because it is possible for a mutation to exhibit altered behavior in only a small subset of cases, and it is infeasible to test all possible combinations of test parameters. Mutations have control over a wide variety of behavior, such as modifying memory access or register reads. Our framework keeps track of killed and surviving mutations to calculate the final mutation score.

Table I: Instructions covered

| Instruction Group | Instructions |
|---|---|
| Arithmetic | `add`, `addi`, `sub` |
| Relational | `slt`, `sltu`, `slti`, `sltiu` |
| Bit Logic | `xori`, `ori`, `andi`, `xor`, `or`, `and` |
| Bit Shift | `sll`, `srl`, `sra`, `slli`, `srli`, `srai` |
| Memory Access | `sb`, `sh`, `sw`, `lb`, `lh`, `lw`, `lbu`, `lhu` |
| Jump/Control | `jal`, `jalr`, `lui`, `auipc` |
| Branching | `beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu` |

In the ISS we provide several features that are helpful in implementing MTCs. One is that instructions are placed into a queue and executed in the order being queued, regardless of jumps or branches. This differs from the usual way flow control would work in code, where instructions after branches or jumps are possibly skipped and where execution continues depends on the addresses at which instructions are located. We are going to dive into details of this aspect in Section III-C, when we demonstrate concrete implementations of a select set of MTCs.

Table I details the instructions we have covered, broken down into groups of semantically similar operations. These comprise 37 of the 40 instructions of the RV32I base instruction set; we have chosen this subset both to manage scope of this work but also because it represents the most basic set of unprivileged computational, load, store, and control-flow instructions that is the foundation for all conforming RISC-V implementations [8][1].

In Section III-B, we will continue with presenting a selection of our MRs for these instructions. We then provide implementation details on the corresponding MTCs (Section III-C). Next, we detail our framework that executes MTCs to kill mutations (Section III-D). Finally, we go into greater detail on those mutations (Section III-E).

[1]We have omitted the 1) `fence` and 2) `ecall`, `ebreak` instructions since 1) is typically implemented as a no-op at the ISS level and 2) might require privileged access, which we do not consider in this work.

### B. MRs for RISC-V

For illustration purposes, we have chosen the following subset of MTCs, at least one from each semantic grouping in Table I. For each one, we are first going to describe the MR used on a general level of abstraction, followed by the MR expressed in a formula based on predicate logic. An expression such as `and(rs3, or(rs1, rs2))` means that the RISC-V `or` instruction is called with two arguments and the result from the destination register is passed into another call of the `and` instruction, using temporary registers if needed.

Section III-C will show how several of these are instantiated into implementations of concrete MTCs that can be executed by our framework.

Some of these relations, such as those for the bitwise operations `or` and `and` below, do not strictly use multiple invocations of the same instruction but multiple calls to related instructions. They are metamorphic in the sense that these instructions are implemented in the same ISS, often with code shared between these instructions, so we have chosen to include relations like these.

Each of the following MR definitions consists of three parts: the name, a description and a mathematical representation. The name encodes the instruction group of Table I and a short identifier separated by a colon.

**Arithmetic: Associativity** As with arithmetic addition, the order in which several additions are executed is irrelevant to the result of `add` or `addi`. Any valid placement of parentheses in the expression $(a+b)+c = a+(b+c)$ yields the same result. This should also hold true in cases of overflow.

$$\texttt{add}(\texttt{add}(\texttt{a},\texttt{b}),\texttt{c}) = \texttt{add}(\texttt{a},\texttt{add}(\texttt{b},\texttt{c}))$$

**Arithmetic: Commutativity** As with arithmetic addition, the order of operands is irrelevant to the result of `add` and `addi`. An implementation of this relation can later be found in Listing 1.

$$\texttt{add}(\texttt{a},\texttt{b}) = \texttt{add}(\texttt{b},\texttt{a})$$

**Relational: Asymmetry** When exchanging operands of an asymmetric operator, only one of the two expressions can be true. This means that, if a relation $R$ relates $a$ to $b$ it does not relate $b$ to $a$. An example is the less-than operator $<$, as only one of $a < b$ and $b < a$ can be true. As `slt` (set less than) represents a less-than test, we can apply it to this instruction.

$$\texttt{slt}(\texttt{a},\texttt{b}) \implies \neg\texttt{slt}(\texttt{a},\texttt{b})$$

**Arithmetic: Triangle Inequality** For `sltiu` we base the MR on its relational asymmetry. The less-than-or-equal operator $\leq$ is neither asymmetric nor symmetric. This operator is antisymmetric, which means that its result with operands reversed can only be true in both cases when the operands are equal: both $a \leq b$ and $b \leq a$ is true iff $a = b$. However, the same applies to the $\geq$ operator. A mutation that exchanges those two operators therefore cannot be killed through asymmetry alone.

Therefore, we introduce the triangle inequality as an MR. For two numbers $a$ and $b$, either of which can be negative or positive, the inequality $|a|+|b| \geq |a+b|$ must hold, where $|x|$ denotes the absolute value of an number $x$. We demonstrate in Section III-D how this MR is using two invocations of `add` to kill a mutation of `bge`, which performs a greater-than-or-equal test to determine whether to carry out a jump. An implementation is later shown in Listing 2.

$$\texttt{add}(|\texttt{a}|,|\texttt{b}|) \geq |\texttt{add}(\texttt{a},\texttt{b})|$$

**Bit Logic: De Morgan's Law** Using De Morgan's laws, one can represent bitwise `and` using inversions and bitwise `or`.

$$\texttt{and}(\texttt{a},\texttt{b}) = \sim(\texttt{or}(\sim\texttt{a},\sim\texttt{b}))$$

**Bit Shift: Additivity of Shift**: Similar to addition, shifting bits left or right has an additive effect on the total amount of shift, as long as the total number of bit shifts does not exceed the type's width in bits. For example, if one shifts the bits of an integer `a` to the left by one position twice, that has the same effect as shifting them left by two positions once. In the following, this is generalized to bit shifts by `s`.

$$\texttt{sll}(\texttt{a},\texttt{s}+1) = \texttt{slli}(\texttt{sll}(\texttt{a},\texttt{s}),1)$$

**Memory Access: Composition of Half-Words from Bytes** Loading a half-word from a memory location is the same as loading the two bytes that constitute the half-word from its two adjacent memory locations `addr` and `addr+1` in two separate load instructions of one byte each, using load byte unsigned (`lbu`). An implementation is later shown in Listing 3.

$$\texttt{lh}(\texttt{addr}) = \texttt{lbu}(\texttt{addr})|(\texttt{lbu}(\texttt{addr}+1) << 8)$$

**Jump/Control: Cancellation of Opposite Jumps** Jump instructions can use both positive and negative relative offsets. The first will jump forward, the second backward. If one jumps forward by $n$ instructions (offset `x`) and then backward by $n$ instruction (offset `-x`), we expect the net effect on the program counter (`pc`) of the two jumps to cancel out. An implementation of this relation is later shown in Listing 4.

$$\texttt{pc}_1 := \texttt{pc};\texttt{jal}(\texttt{x});\texttt{jal}(-\texttt{x});\texttt{pc}_2 := \texttt{pc} \implies \texttt{pc}_1 = \texttt{pc}_2$$

**Branching: Mutual Exclusivity of Branch Types** The two branching instructions `bne` (branch not equal) and `beq` (branch equal) are opposites of each other in so far that the branch condition of one can only be true if the branch condition of the other is false.

$$\texttt{bne}(\texttt{a},\texttt{b}) \implies \neg\texttt{beq}(\texttt{a},\texttt{b})$$

### C. MTC Implementation

The following listings show the implementations of four of the MRs into MTCs that we have introduced in Section III-B and felt were particularly illustrative. We will start with a short introduction of our test framework and then proceed to explain each code listing.

*1) Preliminaries of Metamorphic Framework:* Both our ISS and the MTCs are written in C++. We have simplified the code presented by omitting most implementation details of our MT framework and focus on the MTCs. An important characteristic of our ISS is that the instructions being called, such as `iss.add()`, are always executed in the order they are called in C++, regardless of jumps or branches. This allows us to implement an MTC for a code flow instruction like `jal` in a manner that is a much more direct translation of an MR. For example, in Listing 4 we can execute two consecutive calls to `jal` without the need for jump labels or having to lay out instructions in memory at addresses that depend on the jump offsets passed to `jal`. If one were to implement the same MTC in assembler, this aspect would complicate the code substantially and make it harder to vary the offset to `jal` for testing. Similarly, if one wanted to test a scenario in which a `jal` instruction jumps to a second `jal`, which then jumps back to the first `jal`, one would produce an infinite loop in conventional assembler and thereby a non-terminating MTC.

*2) Implementation of MTCs:* In the following listings, we adopt the following conventions. The method `check` represents an assertion of the MTC. The first time any assertion evaluates to false during execution, the MTC fails and is aborted. If all executed assertions evaluate to true and the MTC runs to its completion the MTC passes. The listings are written in C++ and `iss` is assumed to be an instance of a class representing our ISS. That class provides attributes to query execution state of the ISS and methods through which instructions can be queued for execution. Instructions currently queued are executed and dequeued whenever the method `run`

is called, in the order the methods for each instruction had been called, until the queue is empty. For example `iss.li(rd1, x)` means that the RISC-V `li` instruction is queued and that it will store the value `x` into register `rd1` when executed. In this case, `x` is what is called an immediate, i.e. a literal value that is passed to instruction without the use of a register. Per convention, instructions that end in `i` have immediates as final parameters. `iss[idx]` denotes the value of the register with index `idx`.

Our framework randomizes register numbers and immediate values, so it is implied in the code listings shown that these can take on any value from the type's domain during execution, unless additional constraints on these are stated. The order these instruction methods take parameters follows the convention of destination registers being the first parameters, with source registers and other parameters following. The listings implement the equations introduced in Section III-B of the respective MR, in the same order. Each listing is followed by a short explanation of the code.

**Listing 1** Commutativity of addition

```
1 iss.li(rs1, a);
2 iss.li(rs2, b);
3 iss.add(rd1, rs1, rs2);
4 iss.add(rd2, rs2, rs1);
5 iss.run();
6 check(iss[rd1] == iss[rd2]);
```

In Listing 1, commutativity of the `add` instruction is tested by passing it the same arguments, in different order, and checking that the results written to registers `rd1` and `rd2` are the same in both cases.

**Listing 2** Triangle inequality

```
1 iss.li(rs1, abs(a));
2 addi(rs1, rs1, abs(b));
3 iss.li(rs2, abs(a + b));
4 iss.jal(rd1, 0); // save pc before.
5 iss.bge(rs1, rs2, off);
6 iss.jal(rd2, 0); // save pc after.
7 iss.run();
8 check(iss[rd2] == iss[rd1] + off);
```

The code in Listing 2 tests, if the `bge` instruction obeys the triangle inequality introduced in Section III-B. For this, the addition formula of the absolute values explained there is performed on the test parameters `a` and `b`. The register `rs1` stores the result of the left side and the register `rs2` that of the right side. As the purpose of this test is to check whether the `bge` instruction branches when that relation is satisfied, we must implement a way to see if a jump has been performed. That can be done by saving the program counter before the branch, into `rd1`, and after the branch, into `rd2`, and then checking how much the values of the saved program counters differ. When a branch has jumped, we expect the program counter to increment by the amount passed as the offset. When the branch has not jumped, we expect execution to proceed with the next instruction. As the program counter increments by four on every instruction, we assume the offset to be greater than four to be able to distinguish these two cases.

**Listing 3** Two byte loads can replace a half word load

```
1  iss.li(rs2, val);
2  iss.li(rs1, addr);
3  iss.sh(rs2, rs1, 0);
4  iss.li(rs1, addr);
5  iss.lbu(rd1, rs1, 0);
6  iss.li(rs2, addr + 1);
7  iss.lbu(rd2, rs2, 0);
8  iss.lh(rd3, rs1, 0);
9  iss.run();
10 int half = (((iss[rd2]&0xff)<<8) | (iss[rd1]&0xff));
11 check(iss[rd3] == half && half == val);
```

In Listing 3, a half-word is stored at memory location `addr` with the store-half instruction `sh`. It is then read back in two pieces of a byte each with `lbu`. When one combines the two bytes back into one half-word, with the bit expression passed to `check`, the expectation is that the resulting value is the same as the one initially written with `sh`.

**Listing 4** Negated offset jumps equal, reverse distance

```
1 iss.jal(rd1, 0); // save pc before
2 iss.jal(x0, off);
3 iss.jal(x0, -off);
4 iss.jal(rd2, 0); // save pc after
5 iss.run();
6 check(iss[rd1] == iss[rd2]);
```

Listing 4 shows how the following MR for the `jal` instruction can be implemented particularly elegantly. A jump by `off` and one by `-off` is performed. In RISC-V, with a pair of two consecutive jumps the first would skip execution over the second jump, so this relation could not be implemented in this manner. With our test framework and our ISS, we can perform two such consecutive jumps and observe that the program counter is incremented by the first jump by the same amount that it is decremented by the second jump. Therefore, we test that the program counter before these jumps, stored in `rd1`, is the same as after these jumps, stored in `rd2`. The parameter `x0` designates the register hardwired to zero, which means that we discard the value returned by the jump instruction, as we are only interested in the resulting program counter.

### *D. MT Framework at the Instruction Level*

In this section we explain how a concrete mutation is killed by an MTC in our MT framework. Listing 5 shows the simplified code of a mutation of the `bge` instruction. The method `apply` contains the mutated implementation of the instruction that is called by the ISS whenever it encounters an instance of `bge` in an MTC and changes the unmodified, correct behavior of the instruction in the ISS to produce a mutant. This mutated version is substituted for the correct version, as long as the mutation is enabled. When the mutation is not enabled by the MT framework, the unmodified, presumably correct version of the ISS is executed. The mutation framework passes the register indices `rs1`, `rs2`, and the jump offset `off` into the `apply` method so that the mutant can base its behavior on those parameters. A mutation can modify other aspects of the ISS, such as the program counter `pc`, as shown on Line 4. On Line 3, the $\geq$ operator (correct behavior) has been replaced by the $\leq$ operator (mutated behavior). That replacement is an example of a mutation and the modified version is a mutant.

**Listing 5** Mutation of the bge instruction

```
1 class BgeMutation : OperationMutation {
2   void apply(rs1, rs2, off) {
3     if (rs1 <= rs2) // Correct: rs1 >= rs2.
4       iss.pc = iss.last_pc + off;
5 }};
```

In order for a mutant to be killed, some MTC must eventually detect that the behavior observed while a mutation is enabled differs from the expected behavior. Whether a change in behavior is detected depends primarily on the MR that is implemented by an MTC. There can be several MRs for an instruction but not every MR can be used in an MTC to kill every possible mutation of that instruction. For example, for an instruction like `bge`, which is supposed to branch iff $a \geq b$, one could test that the behavior of `bge` is consistent with relational antisymmetry. Antisymmetry means that iff $a \geq b$ and $b \geq a$ hold $a$ and $b$ must be equal. One can apply this to `bge` by checking that $a$ and $b$ are equal iff both

`bge(a,b)` and `bge(b,a)` branch. This is a useful MR that, as part of an MTC, succeeds in killing many possible mutations, but that MTC alone fails to discover a mutation of `bge` that replaces $\geq$ with $\leq$, because both operators are antisymmetric. To distinguish the two, one needs a relation that applies to $\leq$ but not to $\geq$. One such relation is the triangle inequality $|a|+|b| \geq |a+b|$. Listing 2 shows an MTC using this relation and the following shows a concrete example in which this is useful for killing a mutation.

During execution, the MT framework randomizes values of test parameters such as instruction arguments, source and destination registers, and others. This is done each time an MTC is executed so that different invocations produce a different set of arguments. It may take many executions of an MTC until a randomized assignment is chosen that ultimately kills a mutation. For example, consider the assignment of $a = 1$ and $b = 2$ in the above relation, which would result in the evaluation of $|1|+|2| \leq |1+2|$ in the mutation, instead of $|1|+|2| \geq |1+2|$ in the correct version. Both are true and therefore this particular assignment fails to kill the mutation. Only choosing a positive and negative value, both unequal to zero, catches the mutation: $|-1|+|2| \not\leq |2-1|$. A mutation can produce behavior that deviates from the unmodified behavior for only a small subset of possible arguments.

Continuing the example, assign $a = -1$ and $b = 2$ in Listing 2. While the MT framework executes the MTC, it will substitute the mutation of `bge` on Line 5 and pass our assignments of `a` and `b` to the mutant. This means that on Line 3 of the `apply` method the expression of the if-branch condition now evaluates to `false`, which results in the `bge` instruction not jumping. Consequently, the assertion on Line 8 fails the MTC. Because the MTC fails, the MT framework considers the current mutation killed, disables it, and enables the next mutation.

### E. Mutation Classes Tailored for RISC-V

Table II details the mutations we have used, broken down by mutation classes. When an operator in one of these classes is found in code, mutations are generated by replacing the original operator with each of the other operators of the same class, with each such replacement producing a mutant. For example, encountering $<$ will generate mutants in which that operator is replaced with $<$, $\geq$, $\leq$, $=$ or $\neq$.

We have chosen these kinds of mutations with view of the operations and expressions that are common in the instructions being tested. In other instances, mutation classes different from ours might be more suitable. For example, many of the mutation classes presented in Jia et al. [17], such as array accesses, do not apply in our case, while replacing immediate types, as shown in Table II, represents a clear opportunity for mutation in the case of RISC-V.

## IV. Evaluation

We have implemented our proposed MT approach for processor verification using RISC-V as a case study. As foundation for our framework we leverage the ISS from the open source RISC-V VP available at GitHub [18]. We extended the ISS accordingly to enable execution of MTCs as described in Section III. In total we defined 95 MRs with corresponding implemented MTCs. For evaluation purposes we utilized the 390 mutations as described in Section III-E.

In the following, we first provide more details on the experiment setting and present general results (Section IV-A), then we discuss the most effective MTCs in more detail (Section IV-B). A full list of MTCs will be available as part of our open source implementation on GitHub (link will follow in the final version).

Table II: Mutation classes

| Mutation Group | Operators | Count |
|---|---|---|
| Arithmetic | $+$, $-$ | 20 |
| Bit Logic | `^`, `&`, `\|` | 14 |
| Bit Shifts | `<<`, `>>` | 6 |
| Relational | $>$, $<$, $\geq$, $\leq$, $=$, $\neq$ | 50 |
| Immediates | `B`, `I`, `J`, `U`, `S` | 96 |
| Source / Destination Registers | `rs1`, `rs2` / `rd` | 162 |
| Current / Previous Program Counter | `pc` / `last_pc` | 10 |
| Memory Load | `(u)byte`, `(u)half`, `word` | 20 |
| Memory Store | `byte`, `half`, `word` | 6 |
| Sign Conversion | `uint32_t`, `int32_t` | 6 |
| Total | | 390 |

### A. Experiment Setup and General Results

Important aspects of our MT framework, such as selection of MTC, test parameters and execution order, are randomized. For our experiments, we use the commonly employed practice of initializing the pseudo-random number generator of the C++ standard library by setting a seed based on system time. We have then evaluated the results of executing the full suite of mutations and MTCs ten times. In the following, we present typical execution results and quantify their variability.

All of the 390 mutations are caught by our MTCs, yielding a mutation score of 100%. We achieved a 100% branch coverage of the instructions we covered, as measured by Gcov. The number of mutations killed, per MTC, had a mean of 4.1 (stdev = 4.3) and a median of 3. We observed that 17 MTCs did not kill any mutations. Those MTCs were found to be redundant in combination with the other MTCs, but running them in isolation revealed that they, too, kill a number of mutations. On average, it took 3807 iterations to kill each mutation (stdev = 38470), with a median of 27 iterations. This shows a great degree of variability in the number of MTC executions it takes and thus the difficulty of killing a mutation.

To contain the scope of this work, we did not assess functional coverage of specification requirements or compare coverage to other testing techniques.

Total runtime of the MT framework with all mutations and MTCs was about 35 seconds, on an Intel® Core™ i7-10510U@1.80GHz system. Execution has been observed to take significantly longer in cases when mutations exist that are not killed by any MTC. However, this was only observable during the iterative process of implementing our MTCs, when there were surviving mutations, not in the final evaluation. In the case of no surviving mutations, the total runtime scales approximately linearly with the number of mutations.

The result demonstrate that MT is a suitable technique for processor verification. In the following, we present the most effective MTCs in more detail.

### B. Details on most Effective MTCs

As there are too many MTCs to cover all of them here, we have ranked them by the number of mutations killed and present the eight highest ranking MTCs from a run (Figure 2). Each bar designates the number of killed mutations and is labeled with the name of the MTC, with the number of mutations repeated to the left. We then go over these MTCs by presenting the underlying MR in the same manner we did in Section III-B, each followed by an equation of propositional logic summarizing the relation, except for the ones we have already covered there.

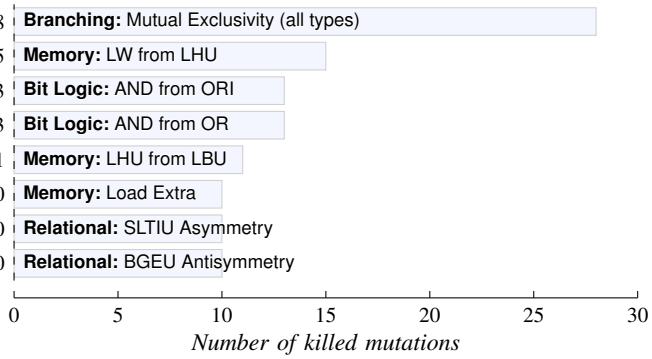


Figure 2: Top 8 Mutation Killers

In Figure 2 we can see that the MTC testing for mutual exclusivity of branches kills the most mutations in that test run, with a total of 28. In our evaluation, we have found that the effectiveness of that particular MTC is in large part due to the number of instructions covered, as it relates five different instructions.

**Branching: Mutual Exclusivity (all types)** The results of branching instructions, given the same operands, are mutually exclusive. This is similar to the complementary behavior between `beq` and `bne` shown in Section III-B, but this one relates all available branch types. This relation states that, if both `beq` and `bge` branch for a given set of operands, then none of `bne`, `blt`, nor `bltu` will.

$$\texttt{beq} \vee \texttt{bge} \implies \neg(\texttt{bne} \vee \texttt{blt} \vee \texttt{bltu})$$

**Memory: Load Extra** Storing a value to a memory location at offset `m` will leave the data in adjacent memory locations unchanged. Therefore, we expect that one can read back values `x1` and `x2` with `lb` from the surrounding addresses that had been written there with `sb` previously.

$$\begin{aligned}&\texttt{sb}(\texttt{m}-1,\texttt{x1});\ \texttt{sb}(\texttt{m}+1,\texttt{x2});\ \texttt{sb}(\texttt{m},\texttt{x3})\\ \implies &(\texttt{lb}(\texttt{m}-1)=\texttt{x1}) \wedge (\texttt{lb}(\texttt{m}+1)=\texttt{x2})\end{aligned}$$

**Bit Logic: AND from OR/I** A bitwise `or` can be expressed with bitwise `and` using De Morgan's law, as shown in Section III-B. One version covers instruction with register operands, the other the versions taking immediate operands.

**Memory: LW from LHU / LHU from LBU** An invocation of `lh` can be constructed from combining the results of two `lbu` instructions, from adjacent memory locations, as shown in Section III-B. Similarly, two invocations of `lhu` can be used to replace one instance of `lw`, by shifting the upper half-word to the left and combining it with the lower half-word.

$$\begin{aligned}\texttt{lw}(\texttt{off}) &= (\texttt{lhu}(\texttt{off}+2) << 16) \mid \texttt{lhu}(\texttt{off})\\ \texttt{lhu}(\texttt{off}) &= (\texttt{lbu}(\texttt{off}+1) << 8) \mid \texttt{lbu}(\texttt{off})\end{aligned}$$

**Memory: Load Extra** Loading from memory after storing a value in that location sets extraneous bits of the target register to zero. The other bits match the originally written value.

$$\texttt{sb}(\texttt{x},\texttt{off}) \implies (\texttt{lbu}(\texttt{off}) \,\&\, \texttt{0xff}) = \texttt{x}$$

**Relational: SLTIU Asymmetry** As in Section III-B, expresses the asymmetry of the $<$ relation implemented by the `sltiu` instruction.

**Relational: BGEU Antisymmetry** This relation is based on the same type of antisymmetry explained in Section III-B. When exchanging the operands of two invocations of `bgeu`, both instances can only branch if the operands $a$ and $b$ are equal. That is the only case in which both $a \geq b$ and $b \geq a$ can be true.

$$\texttt{bgeu}(\texttt{a},\texttt{b}) \wedge \texttt{bgeu}(\texttt{b},\texttt{a}) \implies \texttt{a} = \texttt{b}$$

It can be observed that these presented MRs cover a large set of different instruction groups, including memory, branching and computational operations.

## V. Conclusion and Future Work

In this paper we proposed MT for processor verification using RISC-V as a case study. Therefore, we came-up with a novel set of MRs and implemented corresponding MTCs tailored for RISC-V. Our experimental mutation-based analysis, using an efficient on-the-fly MT framework at the instruction-level, demonstrated the effectiveness of the MRs to kill all mutations, which shows that MT is a viable technique for the domain of processor verification. Based on our MR's generality and the universality of the instruction set chosen, we expect our MRs to be of use for many other processors. Separate investigations will need to be undertaken to show how MT can best complement other test approaches.

For future work we plan to:

- Leverage our MRs for processor verification at the register-transfer level.
- Explore application of MT for different ISAs or RISC-V instruction set extensions.
- Investigate formal techniques to enable automated generation of MRs.

## References

[1] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," vol. 42, no. 3, pp. 264–284, 2015, publisher: IEEE.

[2] S. Segura and Z. Q. Zhou, "Metamorphic testing 20 years later: A hands-on introduction," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 538–539.

[3] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," vol. 49, no. 6, pp. 216–226, 2014, publisher: ACM New York, NY, USA.

[4] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," vol. 62, no. 3, pp. 61–67, 2019, publisher: ACM New York, NY, USA.

[5] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," vol. 42, no. 9, pp. 805–824, 2016, publisher: IEEE.

[6] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" vol. 40, no. 1, pp. 4–22, 2013, publisher: IEEE.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," vol. 41, no. 5, pp. 507–525, 2015-05.

[8] RISC-V Foundation, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, A. Waterman and K. Asanović, Eds., 2019.

[9] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, A. Waterman and K. Asanović, Eds., 2019.

[10] M. Hassan, D. Große, and R. Drechsler, "System-level verification of linear and non-linear behaviors of RF amplifiers using metamorphic relations." ASP-DAC, 2021, pp. 761–766.

[11] J. Liu, "Metamorphic testing and its application on hardware fault-tolerance," 2011, ECE Project Report at University of Wisconsin.

[12] "RISCV-DV," https://github.com/google/riscv-dv.

[13] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.

[14] "RISC-V torture test generator," https://github.com/ucb-bar/riscv-torture.

[15] "OneSpin 360 DV RISC-V Verification App," https://www.onespin.com/solutions/risc-v, 2020.

[16] "RISC-V formal verification framework," https://github.com/SymbioticEDA/riscv-formal.

[17] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," vol. 37, no. 5, pp. 649–678, 2011-09.

[18] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp.