Identification of ISA-Level Mutation-Classes for Qualification of RISC-V Formal Verification

Milan Funck[1] Sallar Ahmadi-Pour[2] Vladimir Herdt[1,2] Rolf Drechsler[1,2]

[1]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
milan.funck@dfki.de
[2]Institute of Computer Science, University of Bremen, Bremen, Germany
{sallar, vherdt, drechsler}@uni-bremen.de

***Abstract*—RISC-V has generated a lot of interest in academia and industry alike due to the open source, modular, and royalty-free design of the Instruction Set Architecture (ISA). With its modular extensibility and the ability to customize the ISA to meet application-specific needs, new challenges arise in terms of verification. Various approaches have been proposed to overcome these challenges, including traditional simulation-based verification and formal verification. While formal verification is considered thorough, its quality heavily depends on the properties and assumptions of the proofs meeting the exact specification.**

In this paper, we propose a structured and mutation-based approach for qualifying formal verification techniques related to the RISC-V ISA. Specifically, we identify rules and mutation classes that can be used to derive a set of mutations to test the capabilities of formal verification tools. We evaluate our approach through a case study, applying a set of mutations to an open-source RISC-V processor, which we verify using the riscv-formal formal verification framework. Our results identify verification gaps uncovered through the generated mutations. We discuss the impact of these identified gaps and how they can be assessed within the context of formal verification.

I. Introduction

RISC-V [1], [2] is a modern, open source and royalty-free *Instruction Set Architecture* (ISA) that gained significant momentum in academia and industry. Besides its free and open nature, a key feature that enabled the ongoing success story of RISC-V is the highly modular design with a broad range of configuration options. The foundation of RISC-V is the mandatory base integer instruction set which is available with different register widths, including 32 and 64 bit architectures. On top of that, a set of optional standard instruction set extensions, such as multiplication and division, are provided. Moreover, custom instruction set extensions can be integrated to build highly application specific solutions.

While the extensive modularity brings many benefits in building an efficient processor design, from the verification perspective new challenges arise and more effort is required. Beside verification of the instruction set extensions, it is also important to verify that the behavior of the base integer instruction set still behaves correctly in combination with new extensions. Different approaches have been developed to address the verification problem, the official directed test suites being the first to mention. This includes the RISC-V unit tests from the University of Berkeley [3] and the RISC-V architectural tests developed by a dedicated RISC-V task group [4]. For a more comprehensive functional verification, a set of RISC-V assembly test generation techniques have been investigated. They leverage predefined templates with randomization [5], fuzzing-based techniques [6] as well as specification-driven generation [7]. A modern and generic framework in this regard is the RISC-V DV [8] verification framework from Google. It utilizes a constraint-based test generation approach and provides generic interfaces to integrate a high-level functional reference simulator in combination with the RTL processor under test.

Although, due to their ease of use and scalability, such simulation-based test generation techniques are an important part of the overall verification effort, they are necessarily incomplete with regard to the verification coverage. Thus, formal verification techniques have been developed for RISC-V. A popular and freely available formal verification tool for RISC-V is the *riscv-formal* [9] framework. It leverages model checking techniques and reasons about the processor behavior by means of assumptions and assertions. The processor behavior is observed through a dedicated *RISC-V Formal Interface* (RVFI), which is a requirement to apply riscv-formal. The RVFI has already been implemented by several RISC-V processors, such as PicoRV32 [10] and VexRiscv [11]. Based on riscv-formal several intricate bugs have been found such as erroneously not identifying reserved compressed instructions as being illegal or an incorrect processing of a jump bit mask [12].

While the found bugs already demonstrate the effectiveness in bug hunting, further analysis is required in order to reason about the comprehensiveness of the formal verification tool. A common approach is to utilize a mutation-based evaluation. Each mutant represents a defect in the processor under test. A mutant is said to be killed, if it is detected by the verification tool. However, a mutation-based evaluation requires the

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, ECXL under grant no. 01IW22002 and VE-HEP under grant no. 16KIS1342.

979-8-3503-0737-5/23/$31.00 2023 IEEE

availability of a strong set of mutants.

For a set of mutations targeting formal methods, the assessment of both the strength and the completeness needs to differ significantly from the corresponding assessment for traditional simulation-based testing strategies. The biggest difference is that if a formal tool detects a certain type of error, it is likely to detect all other similar errors as well. As a result, any kind of simulation-based coverage metrics cannot be applied in this case. Instead, the evaluation needs to focus on whether the formal tool actually covers all aspects of the given specification, for which an adapted mutation-based approach is highly suitable.

Contribution: Therefore, in this paper we propose a dedicated methodology to identify a strong set of mutation-classes at the ISA-level tailored for RISC-V and designed specifically for formal verification approaches. Each mutation-class covers a different aspect of the ISA specification and represents a set of related mutants. We consider positive (the expected results are provided) as well as negative (nothing beyond that happens) aspects. Negative aspects are in particular important to check that the formal verification tool does not miss important use-cases in its property set. In our methodology we propose a structured approach that relies on a set of dedicated rules to derive the mutation-classes by manual inspection of the RISC-V ISA. We focus on identification of corner case scenarios, since basic sanity and functional checks are already well covered through existing test generation techniques. For example, due to the extensive modularity of the RISC-V ISA corner cases in the specification arise at the intersection points of the different RISC-V instruction sets. Based on the identified mutation-classes, we derive a set of representative ISA-level mutants that cover the mutation-classes, and are designed to assess the capabilities of formal verification to avoid unintentional gaps in the verification effort. For evaluation purposes we leverage the RISC-V processor of the open source MicroRV32 platform [13] which already provides the respective RVFI interface required by riscv-formal and is available at GitHub. We integrate each mutant, one after another, in the RTL description of the MicroRV32 processor and check if riscv-formal is able to kill the mutant. Our evaluation revealed several holes in riscv-formal which demonstrate the effectiveness of our approach in identifying strong ISA-level mutation-classes tailored for RISC-V. Moreover, our results enable to strengthen the freely available riscv-formal tool to contribute towards building a complete formal verification solution for RISC-V.

II. Related Work

In addition to the already covered RISC-V test generation techniques, we briefly review formal verification techniques alongside the already introduced riscv-formal. One approach is pursued by *OneSpin 360 DV* for RISC-V [14], which enables model checking techniques to obtain a complete proof and also covers security aspects. Another formal verification tool for RISC-V is provided by *Axiomize* [15], [16], which also relies on formal property sets. Both tools show very strong results but are proprietary. Besides model checking, an alternative approach for RISC-V formal verification relies on theorem proving techniques by formalizing the RISC-V ISA in Kami [17]. Recently, a research approach for the verification of pipelined micro-architectures has been presented, which leverages symbolic QED techniques tailored for RISC-V, called *C-S2QED* [18]. We decided to use riscv-formal for our evaluation since it is a very mature and freely available formal verification tool. It has already been integrated with several popular RISC-V processors, further underlining its importance in the RISC-V community.

Mutation-based testing, in general, has a long history in the verification domain, going back to the software domain and expanding into a broad range of different applications [19]. This includes applications at the system-level using virtual prototypes and embedded systems software [20], as well as the hardware level using fault-injections at the register-transfer level [21]. Mutation-based testing principles have also found their way into commercial tools such as Certitude from Synopsys, which goes back to [22]. The principle has also been applied to guide simulation-based processor verification as a quality metric [23].

Specifically in the RISC-V context, we are aware of two recent mutation-based approaches for evaluating test case coverage. The first approach evaluates the effectiveness of the RISC-V compliance test suite [24], while the second one evaluates a new concept of metamorphic testing (i.e., testing without using an explicit reference model) for finding bugs in an instruction set simulator [25]. Both approaches use standard mutation classes inspired by the software domain (like replacing one register with another or switching immediates), which enables the production of a large set of mutations in an automated way. However, we focus on corner case scenarios at the ISA level instead and aim to evaluate RISC-V formal verification techniques, resulting in a significantly different methodology and obtained mutation classes.

III. Preliminaries

A. RISC-V

At its core, the ISA features a base integer instruction set with either 32, 64, or 128 bits as its standard bit width. Denoted as RV32I, RV64I, and RV128I respectively, optional instruction set extensions such as the M- (multiplication and division), A- (atomics), and C- (compressed) extensions are defined. Furthermore, RISC-Vdefines 32 general-purpose registers `x0` to `x31` (with `x0` being hardwired to constant 0). The bit width of these registers depends on the variant of the base integer instruction set in place (e.g., RV32I uses 32-bit-wide registers). One of the optional instruction set extensions includes *Control and Status Registers* (CSRs), which are added to the base set of general-purpose registers (GPRs) offer control and status registers for hardware timers, meta-information and enables extended processing features such as interrupts, user-modes, and performance counters.

More information on the RISC-V ISA instruction set, together with its extensions and on the privilege architecture

specification, including CSRs, can be found in Volume I [1] and Volume II [2] of the specification, respectively.

B. RISC-V Formal Verification Framework riscv-formal

The RISC-V Formal Verification Framework riscv-formal [9] is a microarchitecture independent set of formal tests (bounded model checks) for RISC-V ISA processors provided and maintained by YosysHQ. As a backend, riscv-formal uses SymbiYosys[1], which is a front-end driver for Yosys-based formal verification of safety, liveness, and reachability properties formulated via assertions, assumptions, and cover statements. It utilizes SMT solver engines to perform either bounded model checks or k-induction. Within riscv-formal, each instruction is covered with its own formal testbench and additional formal tests are available for the liveness of the processor and consistency checks. To automate the verification flow, riscv-formal is accompanied by scripts and is configurable to work independently of the specific processor implementation, as long as the RVFI is implemented and connected to the provided wrapper within riscv-formal. The list of bugs and errors that riscv-formal aims to find includes, but is not limited to, incorrect instruction semantics, incorrect bypassing/forwarding, and errors in the memory interface for STORE and LOAD instructions.

IV. Difference in Challenges between Formal and Non-Formal Verification Approaches

Verification has become an essential and well-integrated part of modern circuit design processes. The ultimate goal is to ensure the complete compliance of a design with its specification.

However, due to the increasing complexity of modern designs, this goal often needs to be adjusted to provide a certain level of confidence assurance. This adjustment is necessary because the number of tested input stimuli is restricted to achieve realistic execution times.

The most commonly used approach seems to be the simulation based verification. Coverage metrics/strategies and mutation-based evaluation are employed to optimize the compromise between feasibility and completeness of coverage. However, formal methods approach verification from a different perspective. In the case of bounded model checking, the design under test and the given specification/property are transformed into a boolean equation, which is then computed by a solver engine. This means that the coverage is always complete in relation to the formulated specification, at least up to a certain depth. Therefore, the question of completeness applies to the formulated specification itself, rather than the exhaustiveness of the considered test vectors or other metrics used in simulation-based approaches.

As design complexity and the comprehensiveness of properties increase, the complexity of the boolean equation to be solved also rises, resulting in exponentially growing solver times. To make the formal proof of a certain specification feasible, it becomes necessary to make certain abstractions, assumptions, and restrictions. In the case of verifying a RISC-V CPU, such an abstraction can be made by reducing its correct functional behavior to its original purpose of sequentially executing instructions with a determinable result (for out-of-order execution, this holds at least for certain sequences of instructions). Regardless of microarchitectural details like pipelining or specific timing aspects, a correctly implemented CPU will always start in a certain state and, depending on the next fetched instructions and other inputs, will reach a specific target state after executing the instruction. The RISC-V Formal Verification Framework is based on such a behavioral abstraction, which is depicted in Fig. 1.

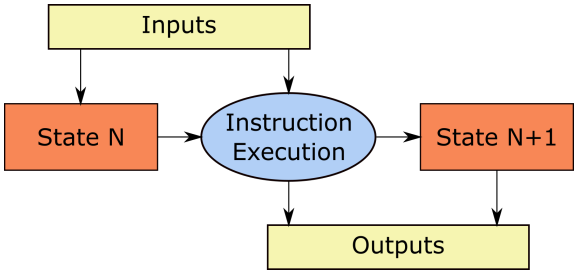

Fig. 1. Abstract model for the processor behavior

The state consists of the Program Counter, the standard Register File, and all registers needed for the CSR-extension, the Floating Point extension, or other extensions that add registers. The inputs and outputs encompass the interaction with memory buses, as well as other possible signals like external interrupts, exceptions, or halt signals.

The problem lies in the fact that certain erroneous or missing behavior may only be active or visible under certain circumstances, which can be systematically (unknowingly) prevented through the abstractions and assumptions made to facilitate the proof. As a result, a formal proof may pass without detecting the error or missing behavior. Thus, a design may pass a formal proof completely, but the proof itself may not fully comply with the given specification. The authors of [26] refer to this as *vacuous satisfaction* and suggest that special sanity checks and different coverage metrics are required to address this increasingly common type of verification gap in formal verification. They propose checking for the completeness of the implemented specification instead.

When considering a mutation-based evaluation of formal verification approaches, it is therefore necessary to include a set of mutation classes that address the abstractions made, as well as the specific challenges and pitfalls of the formal domain. In conclusion, formal verification methods offer a different type of challenge than simulation-based verification methods do, regarding the specification to be checked against. When considering a mutation-based evaluation of formal verification approaches, it is therefore necessary to include a set of mutation classes that address the abstractions made, as well as the specific challenges and pitfalls of the formal domain.

[1]See https://github.com/YosysHQ/sby for more information.

V. ISA-Level Mutation-Classes for Qualification of RISC-V Formal Verification

In this section, we present our proposed approach for identifying ISA-level mutation classes tailored for the qualification of RISC-V verification methodologies. We start with an overview (Section V-A), then describe in more detail the rules we use to identify mutation classes (Section V-B) and finally present our obtained mutation classes (Section V-C).

A. Overview

Fig. 2 shows an overview of our approach. It is separated into two subsequent phases: 1) Preparation (top of Fig. 2) and 2) Evaluation (bottom of Fig. 2).

In the first phase, we apply a structured approach that relies on a set of dedicated rules to derive the mutation classes through manual inspection of the RISC-V ISA. Our approach incorporates expert knowledge and takes the requirements of formal verification into account. Furthermore, we focus on identifying corner case scenarios that might be accidentally overlooked by the formal tool. We provide a more detailed description of our rule-based strategy to identify the mutation classes in the next section (Section V-B). Based on the mutation classes, we sample a single representative mutant from each class. This sample set of mutants effectively covers the mutation classes and enables a more efficient evaluation.

The second phase is the evaluation. Each representative mutant is integrated into the RISC-V processor of the MicroRV32 platform, with modifications made to the processor behavior to exhibit the defect described by the mutant. It is important to note that the mutants pertain to the ISA-level, while the processor itself is implemented at the microarchitecture-level. Therefore, there can be different possible ways to integrate a mutant into the processor's microarchitecture. This observation is particularly important when considering negative testing, which we will discuss further in the subsequent sections. Subsequently, we employ a formal verification approach (in this case riscv-formal) to check if the mutant is "killed," meaning that the integrated defect in the MicroRV32 processor is detected. Based on the evaluation results, we can assess the effectiveness of the formal verification tool in identifying the mutants and evaluate its completeness in the verification process.

In the following sections, we provide a more detailed description of our structured approach to identify the RISC-V ISA-level mutation classes.

B. Identifying Mutation-Classes

As already explained in Section IV, a formal tool will find every fault it observes. The question is if it actually observes every aspect of the given specification. To identify mutation classes in a structured methodology based on this observability, there are three aspects to consider:

1) What faulty behavior does a mutation introduce?
2) Where is this faulty behavior observable?
3) Does the faulty behavior only occur under certain circumstances or conditions?

When applying this to the ISA-Level CPU abstraction (Fig. 1), every violation the RISC-V ISA would be a possible faulty behavior introduced by a mutation. However, a faulty behavior in relation to a certain input can only be observed in either the state (e.g., PC or general-purpose registers) or in the generated output signals (e.g., memory address or fetch commands). Additionally, a faulty behavior could only occur under certain conditions (e.g., specific input values or counter overflows, etc.).

A basic template for this two-dimensional observability space, in which any ISA violation can be placed, is shown in Table I. This classification can be gradually divided into finer subclasses, which we will demonstrate in Section VI. For example, the state faults can be split into several register groups or even the individual registers. The same goes for the output faults and the different conditions under which a fault would appear, creating more and more cells. Certain mutations will fit into multiple cells. To guarantee that a formal tool observes every kind of fault, it is advised to find a set of mutations that addresses every cell individually. For example, if a formal tool only ever observes the register values, it will not find any faulty bus behavior. If a mutation introduces faulty PC behavior, it will show in the PC register and in the generated fetch-bus command. However, such a mutation would not uncover the verification gap concerning bus interactions. This opens up the possibility to introduce certain completeness metrics for a set of mutations, based on how finely the two-dimensional observability classification space is chosen and how well the individual cells are addressed.

As for the faulty behavior introduced by the mutations, every ISA violation would be suitable. However, from the perspective of observability, complemented by a special focus on formal verification and the RISC-V ISA, the choice of mutations can be guided by a few rules, grouping them into sensible categories:

1) *ISA path coverage analysis* that covers different logical paths according to the specification.
2) *Difference-based analysis* focusing on intersection points between different instruction sets.
3) *Special pitfalls* regarding formal verification, covering unintended behavior or side effects.
4) *ISA corner cases* specific to the RISC-V specification that are not covered by the first three rules.

Each rule is designed to cover specific aspects of the functional specification including corner cases, which might be accidentally incorrectly implemented or not considered by the formal verification tool.

The first rule is intuitive as it represents different logical cases in the specification in a general sense. This covers the correct implementation of all instructions. Additionally, more special paths are of interest, such as the handling of illegal instructions and the corresponding traps/exceptions thrown.

As an example of the second rule, consider the RISC-V ADDI instruction. It has a slightly different specification for the 32-bit and 64-bit ISA, resulting in different behavior. A

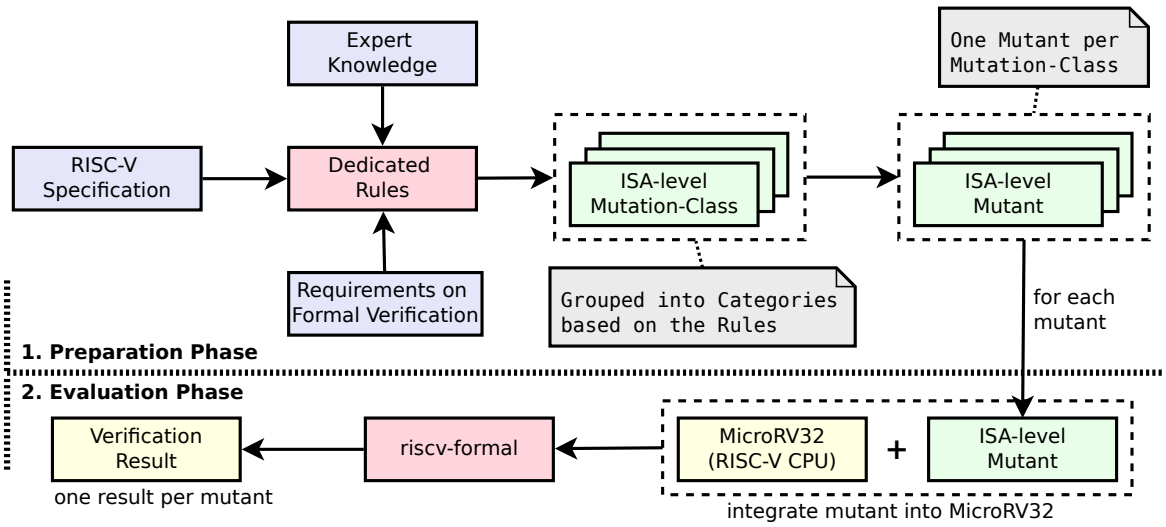

Fig. 2. Overview on our proposed methodology.

64-bit implementation applies sign-extension to the immediate and the 32-bit result (to cover a 64-bit register range). However, when designing a RISC-V processor that supports both the 32-bit and 64-bit instruction set, it can easily be forgotten to implement the necessary differentiation. Such small mistakes could go unnoticed because an application does not generate the respective code, but in other cases, this would lead to wrong results. Based on this observation, problematic and easily overlooked ISA differences can be identified, and particular instructions can be used as representative mutants.

The third rule is a bit more elaborate and designed specifically with formal verification and its possible pitfalls in mind. The idea is to ensure that no additional side effects occur, like generating faulty bus signals in addition to the correct update of the destination register. By considering such cases, we can formulate respective mutation classes to cover these scenarios. Please note that the idea in this context is not to ensure that the verification tool finds that specific mutation, but in a more general sense, it is able to handle this class of errors. Such an error can also manifest itself based on intricate microarchitectural pipeline bugs, which the formal tool should find as well if it has been designed to observe every important aspect of the design under test. Hence, this negative testing rule can be considered to perform sanity checks regarding the completeness of the formal verification tool.

As an example, riscv-formal is a bounded model check based on a special interface that feeds information from a RISC-V CPU to both the formal tool and a reference RISC-V instruction simulator. The tool then compares the results of instructions executed by the device under test with the reference simulation. In this case, there are obvious limitations to this verification approach. One limitation is the presence of faulty behaviors that occur outside the chosen boundary of the model check. Another limitation is cases in which a faulty behavior somehow produces faulty signals on the mentioned interface, causing the reference to base its results on faulty inputs.

TABLE I
BASIC ISA-LEVEL MUTATION CLASSES

	unconditional	**conditional**
state fault	*ISA violation?*	*ISA violation?*
output fault	*ISA violation?*	*ISA violation?*

For the fourth and last rule, we have thoroughly searched the RISC-V ISA for specific peculiarities that can easily be overlooked in the design process of both a CPU design and a corresponding formal verification framework. Such corner cases should, therefore, be included in every set of mutations. One example would be the division by zero, which is handled in a specific way in RISC-V, or the handling of conditional branches.

C. Identified Mutation-Classes

In this section, we describe the identified mutation classes based on the four strategies mentioned in Section V-B. The abstract CPU model (Fig. 1) is taken into account, and the mutation classes applied to the two-dimensional observability categories described in Section IV are intended to systematically guide the verification engineers, forming a set of mutations.

I. **ISA path coverage analysis, that covers different logical paths according to the specification** This strategy focuses on general mistakes and errors during translation/implementation of the ISA into a microarchitecture. In principle, this includes corner cases while the intent is to focus on more general ISA-paths and sanity checks.
   a) Mutations altering the result of an instruction.
   b) Mutations altering the bus interaction.
   c) Mutations that alter the correct pc-behavior.
   d) Mutations that alter the correct behavior concerning traps/interrupts and exceptions.

II. **Difference based analysis, focusing on intersection points between different instruction sets** This strategy is meant to ensure strict distinction between specifications of different RISC-V ISA-extensions.
   a) Differences in instructions
   b) Additional/missing instructions
   c) Architectural differences

III. **Special pitfalls regarding formal verification, covering unintended behavior or side effects** This strategy is based on expert knowledge and experience regarding formal verification. The specific details of the following subclasses are outlined in Section IV.
   a) Vacuous Satisfaction
   b) Unintended behavior
   c) Faulty proof-assumptions and preconditions

IV. **Other RISC-V specific corner cases that are not covered by the first three strategies** The RISC-V ISA contains several design decisions, that lead to certain RISC-V specific corner cases. Especially some non-trivial corner cases are hidden in the details of the ISA specification and should not be neglected. For example:
   - No integer computational instructions cause arithmetic exceptions.
   - Conditional branch instructions only throw exceptions on a taken branch.
   - Loads with a destination of `x0` must still raise any exceptions and cause any other side effects even though the load value is discarded.
   - Register `x0` is hardwired to zero.

VI. Evaluation

In this section we describe our samples of mutations to conduct an exemplary qualification of the riscv-formal verification framework. The set of mutations is then generated using the RISC-V processor of the MicroRV32 platform, which implements the necessary interface and the respective results are discussed afterwards. The experiment was executed with Ubuntu 20.04 LTS on an Intel Xeon Gold 6240 CPU clocked at 2.60 GHz and with 383 GB of system memory.

A. Mutations

In addition to the unchanged version of the MicroRV32 processor, serving as a reference, we chose ten different mutations. We derived each of the mutations from one of the subclasses discussed in Section V-C, only neglecting those, which would require further discussions and overly complicated manipulations to the core. Many of these mutations could arguably be assigned to multiple mutation classes at once. For each mutation, we also cross-reference the respective mutation class it was derived from.

1) The ALU is manipulated, such that the ADD Instruction subtracts instead. (I.a)
2) The PC now is always incremented by 8 instead of 4. (I.c)
3) The zero instruction is now handled as a RI format instruction, instead of being illegal. (I.d)
4) The shift instruction is decoded as specified in the 64I ISA instead of 32I ISA. (II.a)
5) The LWU instruction from the 64I ISA is now legal, working exactly like the LW instruction. (II.b)
6) The decoder is manipulated to leave out the evaluation of the func3 value for all store instructions. (III.a)
7) The instruction results are also written to another predefined register apart from the destination register. (III.b)
8) The `x0` register can now be written to. (IV.)
9) Conditional branches will raise an instruction-address-misaligned exception, even on a branch not taken. (IV.)
10) After a certain amount of cycles, the load values will always be incorrect. (I.b + III.c)

With regard to the observability aspects discussed in Section V-B, this first exemplary set of mutations has also been allocated to the respective cells of Table II. It is an expansion of the basic template shown in Table I, better suited to the chosen set of mutations. The state has been split into the program counter (PC), the general-purpose registers, and the control-and-status registers, with the latter mainly showing thrown exceptions. For the generated output signals, a differentiation has been made between the instruction bus and the memory bus. Furthermore, the conditional faults have been separated into state-dependent and input-dependent ones. Mutations marked with '*' indicate that they will only be observable in the PC or the generated instruction bus outputs if an exception causes the CPU to jump to a trap handler. The table shows that every cell but one is addressed, although most of them not individually, which will be discussed in relation to the results.

B. Results

The results produced by formally verifying the exemplary set of mutations are shown in Table III. The left side of the table shows an identifier of our mutation sample (referring to our list in Section VI-A and the mutation-classes from Section V-C) and a short description of the behavior of the mutation. The right side shows the evaluation result for each mutation and a summary of whether the mutation is killed (found by riscv-formal) or not. Once at least one test of riscv-formal fails, the mutation is considered killed. For reference, we include an unaltered version of the MicroRV32 CPU that passes all the riscv-formal tests.

The results show that while some of the mutations are found by riscv-formal, there are still verification gaps uncovered through our systematic approach of generating mutants. We will discuss some of the results and their respective details further in this section.

The manipulated ADD instruction was realized by making the ALU subtract instead of add for the ADD instruction. This mutant was killed by riscv-formal. Apart from dedicated checks for the PC, each instruction check verifies the PC value. Therefore, the second mutation (I.c) is also killed. Handling the all-zero instruction as an RI-format instruction (I.d) does

TABLE II
SPECIFIC ISA-LEVEL MUTATION CLASSES

		unconditional	**conditional**	
			state-dependent	input-dependent
state fault	PC	2), 3), 4)*, 5)*	9)*	4)*, 9)*
	GPRs	1), 5), 7)	8)	4), 8)
	CSRs	3), 5), 6)	9)	4), 9)
output fault	IBus-interaction	2), 3), 5)*, 6)*	9)*	4)*, 9)*
	MBus-interaction	5), 6)	10)	-

TABLE III
RESULTS OF THE MUTATION-BASED EVALUATION

Mutation	Mutation Class	Description	# PASS	# FAIL	Found mutant?
-		Unmodified (reference)	43	0	-
1)	I.a	ALU subtracts on ADD instruction	31	12	✓
2)	I.c	Program counter increments by 8	6	37	✓
3)	I.d	Decode defined illegal instruction (all zeroes) as legal	43	0	✗
4)	II.a	Decode shift instruction as RV64I shift instruction instead RV32I variant	43	0	✗
5)	II.b	Decode LWU instruction from RV64I when only RV32I is allowed	43	0	✗
6)	III.a	Ignore funct3 field on STORE instructions at decoding	43	0	✗
7)	III.b	Write to register `x30` as well, whenever any register is written	43	0	✗
8)	IV.	Make register `x0` writable	5	38	✓
9)	IV.	Raise trap on misaligned branch target on branch not taken	37	6	✓
10.1)	I.b + III.c	Manipulate loaded values after 100 and 600 loads respectively	43	0	✗

not effectively lead to a different result. However, according to the RISC-V ISA specification, this instruction should be illegal, and the core should raise an exception. The survival of this mutation shows that there are details of the decoding process that are not considered by riscv-formal. This is further underlined by the fact that the mutant for decoding the SHIFT instructions (II.a) is not killed. Tolerating the LWU instruction (II.b) and the missing evaluation of the funct3 field for the STORE instruction (III.a) all survive the set of formal checks.

The mutation altering the behavior of the register file (III.b) is not killed. That is explainable by the fact, that exact register changes are not actually verified by riscv-formal. Only the interactions between the core and the register file are covered by the formal properties. This helps reduce the complexity of the formal proof but is also a good example for assumptions/abstractions, leading to a verification gap. The mutation of the `x0` register is killed (IV.) because it shows when evaluating the correct results of multiple executed instructions. The corner case regarding the exception handling of conditional branches is correctly implemented by riscv-formal, and the corresponding (IV.) mutant is killed.

The last mutations (I.b and III.c), which we included with two different timing variations, are not killed. This is explained by the fact that riscv-formal mainly applies bounded model checking and can only identify such misbehavior up to a certain unrolling depth. We still included this mutation to highlight these kinds of limitations in such an approach.

VII. DISCUSSION

Examining the results, several mutations survive the formal verification process. One could argue that some of the chosen mutants are very specific and unlikely to appear in a real design. But, as previously discussed, these mutations each represent a whole category of possible design flaws and therefore contribute to a certain aspect of the verification process. If one of these mutations survives the verification process, it suggests that the latter is incomplete in certain areas. Especially in the context of formal verification, this leads to the conclusion that not only a certain corner case got past the test suite, but a whole category of ISA violations might remain unchecked.

Of course, a verification approach should always be evaluated in the context of its natural limitations. The register file could easily be verified in a separate and less complex proof, and the timing-based mutation might simply be out of scope for a bounded model check as used within riscv-formal. On the other hand, inaccuracies within the decoder should not slip through a formal verification approach. Decoding instructions is a crucial part of the processor design, and thus false trust in its correctness could leave opportunities for design vulnerabilities like hardware trojans.

Regarding the completeness of the mutation-based evaluation, the exemplary set of mutations is quite small. Table II displays how thoroughly the observability aspects are checked by the chosen set. Although almost every cell is addressed, only a few mutants address any cell individually. Nevertheless, the mutations were selected considering basic knowledge about the functionality of the riscv-formal framework and demonstrate how effective such a small set of mutations can uncover verification gaps. This leads to the question of what results a more extensive evaluation would produce.

Therefore, we suggest achieving full coverage of the rules and the observability space described in Section Section V-B. This would effectively result in a set of mutations that includes every basic ISA path violation, along with other more specific points of interest, applied to each cell of the corresponding observability table individually. Although beyond the scope of this first demonstration, this approach seems feasible.

In general, our systematic mutation-based approach for qualifying formal verification tools for RISC-V and their properties covers a broad range of property types. While all instructions can be checked for their expected functionality, our approach also covers instruction semantics and decoding that should not be executed and instead should cause an exception. As the ISA-Level covers the instruction's semantics and their decoding in terms of an abstracted CPU model (see Fig. 1), microarchitectural details and possible design optimizations require an extended methodology. This is acceptable since black box and gray box formal verification frameworks like riscv-formal are agnostic to microarchitectural details. If microarchitectural details and design optimizations are checked through formal properties, our methodology can be expanded and concretized to guide verification engineers in finding mutations that target such designs. At least the preservation of the core functionality specified by the ISA after an optimization is automatically addressed without any further adaptions. Through certain mutations, we demonstrated that timing issues and similar non-functional bugs can be introduced and found to be undetectable by the formal verification tool. Generally, bugs related to aspects like liveness or deadlocks do not exist at the ISA-Level and depend on the microarchitecture, thus requiring additional knowledge of the hardware implementation. We believe our approach covers a broad range of possible formal properties that can be checked by leveraging the existing ISA-Level abstraction to systematically generate mutations for qualifying formal verification.

VIII. Conclusion and Future Work

In this work we proposed a mutation based qualification approach for formal verification methods for the RISC-V ISA. Specifically, we provide a structured approach relying on a set of dedicated rules and mutation classes to obtain feasible mutations. We evaluate the approach by using a set of representative mutations to check against the riscv-formal framework. By applying the ISA-level mutations to the RISC-V processor of the MicroRV32 platform and checking the mutated processor with riscv-formal, we were able to identify gaps in the verification approach of riscv-formal. The identified gaps of riscv-formal can be considered severe as they relate to core capabilities of processing units (e.g., instruction decoding). Through the help of our qualification approach, such verification gaps can be found and addressed to increase the quality of the verification suite. To further extend our approach we plan to:

- Investigate the possibility of ISA-level coverage metrics in order to assess the completeness of our method.
- Consider automatic/semi-automatic generation of mutation samples through the formalization of our mutation classes in connection with the formal ISA specification.

References

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

[2] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, 2019.

[3] "RISC-V ISA tests," https://github.com/riscv/riscv-tests.

[4] "RISC-V foundation architecture test," https://github.com/riscv-non-isa/riscv-arch-test.

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

[6] Herdt, V., et al, "Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side," in *DAC*, 2020.

[7] Chupilko, M., et al, "Test Program Generator MicroTESK for RISC-V," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2018, pp. 6–11.

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

[9] "RISC-V formal verification framework," https://github.com/YosysHQ/riscv-formal, 2020.

[10] "PicoRV32 - a size-optimized RISC-V cpu," https://github.com/YosysHQ/picorv32.

[11] "A FPGA friendly 32 bit RISC-V CPU implementation," https://github.com/SpinalHDL/VexRiscv.

[12] "Examples of bugs found by riscv-formal," https://github.com/YosysHQ/riscv-formal/blob/master/docs/examplebugs.md.

[13] Ahmadi-Pour, S., et al, "The MicroRV32 framework: An accessible and configurable open source RISC-V cross-level platform for education and research," *Journal of Systems Architecture*, vol. 133, p. 102757, 2022.

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

[15] "Axiomising RISC-V processors through formal verification," https://www.axiomise.com/risc-v-formal-verification/, 2022.

[16] "Democratising formal verification of RISC-V processors," https://riscv.org/wp-content/uploads/2019/12/12.10-16.40-AXIOMISE__RISCV__Final__RISCV__Summit__2019.pdf, 2019.

[17] "Formal specification of RISC-V ISA in kami," https://github.com/sifive/RiscvSpecFormal, 2022.

[18] Devarajegowda, K., et al, "Gap-free processor verification by S2QED and property generation," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 526–531.

[19] Jia, Yue, et al, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, p. 649678, Sep. 2011.

[20] Kuznik, C., et al, "Native Binary Mutation Analysis for Embedded Software and Virtual Prototypes in SystemC," in *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, Dec 2011, pp. 290–291.

[21] Serrestou Y., et al, "Functional Verification of RTL Designs Driven by Mutation Testing Metrics," in *DSD*, 2007, pp. 222–227.

[22] Hampton, M, et al, "Leveraging a Commercial Mutation Analysis Tool For Research," in *MUTATION*, 2007, pp. 203–209.

[23] Xie, T., et al, "Mutation-analysis driven functional verification of a soft microprocessor," in *SoC*, 2012, pp. 283–288.

[24] Herdt, V., et al, "Mutation-based Compliance Testing for RISC-V," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 55–60.

[25] Riese, F., et al, "Metamorphic Testing for Processor Verification: A RISC-V Case Study at the Instruction Level," in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2021, pp. 1–6.

[26] H. Chockler, "Coverage metrics for formal verification," in *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 2003, pp. 111–125.