Single Instruction Isolation for RISC-V Vector Test Failures

Manfred Schlägl
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
manfred.schlaegl@jku.at

Daniel Große
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
daniel.grosse@jku.at

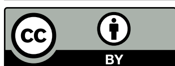
ABSTRACT

Testing complex RISC-V extensions such as *RISC-V Vector* (RVV) with its 600+ highly configurable instructions is crucial. For this reason, test suites have been developed over the last years, including both hand-written and automatically generated tests. Although the process of running these tests is often highly automated, a significant portion of the work, namely the result analysis, has to be conducted manually after the run.

This paper introduces the modular, open-source framework *RVVTS* for positive and negative testing of RVV implementations, featuring a novel technique called *Single Instruction Isolation* with *Code Minimization*, which significantly reduces manual result analysis of failing tests. We demonstrate the effectiveness of *RVVTS* by automatically generating and applying test sets to the *RISC-V VP++ Virtual Prototype* and the *QEMU* emulator, achieving a functional coverage of >94%. For *RISC-V VP++*, our framework detects and minimizes ~1, 849 failures and associate them with 10 isolated, failing instructions. Similarly, for *QEMU*, it detects ~19k failures and relates them to 168 instructions for debugging. Overall, we confirmed 3 new bugs in the *RISC-V VP++* and 2 in *QEMU* (and 7 more are to be analyzed).

1 INTRODUCTION

RISC-V [39, 40], an open standard *Instruction Set Architecture* (ISA), embodies flexibility and scalability, enabling the precise tailoring of processor capabilities to meet diverse application needs without the constraints of unnecessary features inherent in proprietary ISAs. RISC-V supports a range of optional extensions, such as those for floating-point operations, atomic instructions and vector processing, enabling further customization and optimization. Each of these extensions adds a layer of functionality that must be thoroughly tested. The verification process typically involves creating specific test sets that can handle the complexities introduced by these extensions. Although still challenging, testing of simple instruction sets, for example RISC-V base integer, is a well understood problem. For example, the behavior of a simple integer add instruction may only depend on the parameters directly passed to the instruction. The parameter space is manageable and it may be even feasible to hand-craft tests for such instruction sets. The RISC-V compliance test suite also exemplified this, being crafted by hand when it came out [1].



This work is licensed under a Creative Commons Attribution International 4.0 License.

ICCAD '24, October 27–31, 2024, New York, NY, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676755

However, this approach is not feasible any more for two reasons: (i) more comprehensive tests are needed, and (ii) the complexity of the instruction sets increases. Consequently, there has been a push towards the development of automated test generation techniques, also referred as *Instruction Sequence Generator*s (ISGs), to facilitate exhaustive verification processes. Notable methods include the use of constraint solving techniques, guided random generation, quick error detection and equivalent program execution. More details are given in our discussion of related work in Section 2.

Let us now specially look on the complexity challenge introduced by the *RISC-V "V" Vector Extension* (RVV) with its 600+ instructions. RVV allows to efficiently handle data-heavy and parallel processing tasks, making it highly adaptable for advanced applications in machine learning, multimedia, and scientific computing. In contrast to the simple integer add example from above, the behavior of an RVV instruction depends not only on directly passed parameters, but also upon the dynamic configuration and, thus, the architectural state. For example, a RVV add instruction may behave differently not only wrt. the directly passed parameters, but also wrt. the previously set vector length, dynamic type (8, 16, 32, 64 bit), etc. Thus, the parameter space becomes high dimensional, which makes manual test creation no longer efficient. For this reason, dedicated ISGs and pre-generated test sets have been developed over the last years. One example is *RISCV-DV*, which was originally developed by Google [9]. However, this ISG does not support the ratified version 1.0 of RVV. Another example, which overcomes this problem, is *FORCE-RISCV*, maintained by the *OpenHW Group* [5]. It provides an ISG for generating extensive tests and the reference simulator *Handcar* which generates execution traces for these tests. The obtained execution traces can then be compared with traces generated by a comparative run on a *Device Under Test* (DUT). However, a significant portion of work, the analysis of the trace differences, is left to the user. This analysis is largely manual work and involves finding differences, eliminating irrelevant details and isolating instructions, errors and states in a vast amount of traces.

Thus far, we focused on testing with the emphasis on checking that instructions work as expected, called *positive testing*. However, we must also consider possibly unexpected/undesired behavior when the DUT is exposed to invalid instructions. This kind of testing is referred to as *negative testing* [22]. Let us now examine the *Instruction Register* (IR) of a processor which stores the current instruction word to be executed. Then, for negative testing, it is necessary to distinguish between the following cases:

- Invalid instruction word: The instruction word is not defined by any (custom) RISC-V extension.
- Invalid instruction because of unsupported extension: The instruction is specified but not supported by the RISC-V core at hand.

- Invalid instruction because of temporarily disabled extension: For example, the instruction considers floating-point, but floating-point is temporarily disabled (done via the *Control and Status Register* (CSR) *mstatus*).
- Invalid instruction because of dynamic configuration: A good example is the RVV element type set to 8 bit and the current instruction performs a RVV floating-point operation (there is no support for 8 bit floating-point elements).
- Invalid because of parameter(-values): Consider for instance a RISC-V load instruction that is issued with a invalid load address.

As we can see, the number of dimensions in the parameter space increases further, which makes the process of testing even more complex. This has two major implications: (i) the ISGs must be able to generate also invalid state, instruction and parameter combinations in a systematic way, and (ii) the already challenging analysis of the test results becomes much harder due to increased number of parameters and combinations to be considered. We address both challenges in this work.

Contribution: In this paper, we present the modular, open-source framework *RVVTS* for positive and negative testing of RVV, where at the heart is our novel *Single Instruction Isolation* with *Code Minimization* technique. Besides efficient test generation, *RVVTS* allows to reduce manual result analysis of failing tests significantly. The framework supports automation of the full verification chain:

1. grammar-based, coverage-guided ISG,
2. instrumentation and build,
3. measurement of functional coverage,
4. execution on reference simulator and DUT,
5. detection of differences in architectural states (fails),
6. isolation of the failing instruction (*Single Instr. Isolation*) and,
7. creation of minimized failing test case (*Code Minimization*).

In addition, the framework can be used interactively in *Jupyter* notebooks to support the user in tracing causes of detected fails.

In our case studies, we demonstrate the effectiveness of *RVVTS*. We use use the framework to generate ready-to-use test sets for positive and negative testing of RVV implementations for RV32 and RV64, respectively. Each test set achieves a functional coverage of over 94%.

The tests sets are applied on two DUTs implementing RVV in its ratified version 1.0, namely the open-source SystemC [4, 23] TLM based *RISC-V VP++* [33, 35, 36] *Virtual Prototype* (VP) and the open-source *QEMU* emulator [6]. Based on the obtained results, we discuss the differences of positive and negative testing, and demonstrate the *Single Instruction Isolation* with *Code Minimization* technique. We show, for example, that we can automatically minimize about 152k detected fails in the initial version of the VP and associate them with 542 isolated, failing instructions. For both DUTs, we perform a result analysis and discuss several previously undetected bugs. All newly found bugs are reported to the respective open-source projects. *RVVTS*, as well as the pre-generated test sets are available as open-source on GitHub[1].

The paper is structured as follows: First, in Section 2 we discuss related work. Thereafter, in Section 3, we briefly review RVV. In Section 4, we introduce the *RVVTS* framework with a particular focus on the novel technique, namely *Single Instruction Isolation* with *Code Minimization*. The case studies are described in Section 5. Finally, the paper is concluded in Section 6.

[1]https://github.com/ics-jku/RVVTS

2 RELATED WORK

Generating tests for processors is an intensively studied research field. For automation and efficiency, a major step was made by separating the description of the processor architecture from test generation. The approaches in [14, 16] achieved this by employing constraint solving techniques on constraints that capture the architectural description and testing-knowledge. Test generation was further improved in [30] by sharing information between constraints solutions among multiple instructions. To increase coverage, the approach introduced in [17] formed a coverage model via constraints describing execution paths of individual instructions. Also Bayesian networks have been used with the same goal [20]. Furthermore, machine learning approaches emerged, for instance [15, 28] and also fuzzing, e.g. [27, 32]. However, all these approaches do not support RVV, focus only on positive testing, and for failing tests the manual analysis effort to trace down the root cause would be far too high.

Besides this, also RISC-V specific approaches have been developed. One of the first RISC-V ISGs was the *Torture Test* framework [2] based on Scala. RISC-V International hosts the *Architecture Test Special Interest Group* which provides the architectural tests [7]. These tests constitute a basic suite that evaluates critical components of the specification without an in-depth focus on finer details and has been generated using constraints. Another approach was Google's *RISCV-DV* [9], mentioned in Section 1. It utilizes SystemVerilog alongside the *Universal Verification Methodology* (UVM) to continuously generate RISC-V instruction sequences solving constrained-random specifications. However, it does not support RVV 1.0; the basic applicability was nevertheless shown in [29], but the effort for result analysis has already been recognized for positive testing. Recall, that the aforementioned argument is also one major problem for *FORCE-RISCV*. As we will show in the experiments, there is an enormous amount of failed RVV tests if we consider both, positive and negative testing where a manual inspection is practical infeasible (150k+ deviations in our case study, Section 5.2).

Test generation for *Instruction Set Simulator*s (ISSs) leveraging coverage-guided fuzzing has been targeted in [22, 26]. A formal constraint-based specification approach has been presented in [24]. The paper [25] introduced an efficient cross-level testing approach for ISS vs. RTL models which generates an endless instruction stream on-the-fly during simulation. However, all these approaches neither target RVV nor provide solutions for large number of failing test cases.

An entirely different approach is formal verification, as functional behavior is proven. Prominent examples employing model checking techniques are *riscv-formal* [8] and *Siemens EDA's OneSpin tools* [12]. *riscv-formal* does not support RVV, and the *OneSpin tools* are only commercially available. Originally designed exclusively for post-silicon verification, *Quick Error Detection* (QED) has been further extended and used for pre-silicon verification [37]: it is referred to as *Symbolic Quick Error Detection* (SQED), leverages *Bounded Model Checking* (BMC) techniques and has been applied to RISC-V processors [18] and hardware accelerators [38]. The core

idea of SQED is to symbolically check whether the outputs from executing a specific instruction sequence are consistent when the sequence is run twice, provided that the inputs for both executions are identical. The work in [31] introduced EPEX where for a given test program an equivalent test program is generated using SMT. Then, both programs are executed on two instances of the same processor design and the programs have to produce equal architectural states. Although formal methods offer guarantees of correctness, they may suffer from scalability issues. In case of RVV, this problem is even harder as (i) very wide datapath operations typically have to be performed and (ii) due to the dynamic configuration of the RVV instructions.

Finally, there is the formal specification of the RISC-V ISA written in Sail [10]. While the model provides the ultimate truth, currently the RVV part is still under development. If it becomes available, we can easily integrate a reference simulator generated from the Sail model.

3 THE RISC-V "V" Vector Extension (RVV)

Performance of algorithms used in image and video processing, audio manipulation, scientific simulations, and modern AI algorithms can be significantly increased by exploiting *Data-Level Parallelism* (DLP). One way to achieve this are *Single Instruction, Multiple Data* (SIMD) instruction set extensions. SIMD operates by executing identical operations on multiple data elements simultaneously, known as a **Vector**. Initially conceived in the 1970s [21], SIMD gained traction in supercomputing systems pioneered by Cray, evolving into what we now recognize as vector architectures [19].

Processors have included SIMD instructions for about 25 years, categorized as multimedia extensions like Intel MMX, SSE, AVX, or ARM Neon. To simplify *Hardware* (HW) implementation, these *classical SIMD* extensions operate with fixed-size registers (vector length). The vector length and element types are specified in the ISA and encoded directly in instructions. This has implications on scalability, since every change of the vector length means changes in the ISA and previously compiled code cannot take advantage of the larger vectors. In contrast, *vector architectures* such as RVV are a more general. Instead of having a fixed vector length and types encoded in instructions, such architectures have have *dynamic* vectors and *generic* instructions. The vector length and the type of the elements are set *dynamically* with dedicated instructions at runtime. *Software* (SW) can detect and utilize the maximum available vector length provided by the HW at runtime.

The ratified version 1.0 of RVV, specified in [3], adds the following parts to the RISC-V programming model: (i) 32 Vector registers, (ii) 7 CSRs, and (iii) 624 instructions. The length of the RVV registers, `VLEN`, is not fixed in the ISA, but can be chosen by the designer. In addition, registers can be grouped at runtime to increase the maximum available vector size at the expense of a smaller number of usable registers. The 624 instructions can be categorized in configuration, load/store and processing. Configuration instructions, such as `vsetvli` set the *dynamic type* and *vector length*. The extensive set of load/store instructions can efficiently handle even complex data structures (gather/scatter). The processing instructions cover operations on integer, fixed point and floating point data types. In addition, RVV specifies vector reduction (e.g. sum, min, max, ...) and permutation (e.g. move, shift, ...) instructions.

4 THE RVVTS FRAMEWORK

The modular *RVVTS* framework is developed in *Python* and supports integration with *Jupyter* notebooks. This gives the framework the flexibility to support automated processes, such as running long series of tests, as well as semi-automated, interactive use cases, such as tracing specific issues.

4.1 Fundamental Data Structures

A central data structure of *RVVTS* is the *Machine State*, which holds the architectural state relevant to all supported extensions. The *Machine State* holds values of all involved registers (integer, floating-point, vector, and CSR), a trap counter, the last executed *Program Counter* (PC) address and hashes for memory areas. The data structure supports extraction from a RISC-V machine, modification and comparison, and includes a generator for code to bring a RISC-V machine to that state. For example, the framework can execute a program $\mathcal{A}$ on a RISC-V machine and extract the *Machine State*. It then can modify parts (e.g. register value) of the *Machine State*, generate the initialization code, integrate this code in another program $\mathcal{B}$ and execute this program on a RISC-V machine. Program $\mathcal{B}$ would then behave as if it had run immediately after program $\mathcal{A}$, but with the introduced change in the *Machine State*.

Other key data structures are the *Code Block* and the *Code Fragment*, which are used by *RVVTS* to organize and handle assembler code. A *Code Fragment* is a indivisible sequence of at least one assembler instruction implementing an operation. For example a *Code Fragment* for a RVV add operation contains a single vector add assembler instruction. A *Code Fragment* for a *RVV bounded load/store operation* [34] contains multiple assembler instructions to realize the bounding and a vector load/store instruction.

A *Code Block* contains a sequence of *Code Fragments* and forms an entity that can be executed on a RISC-V machine by *RVVTS*. The data structure provides various methods to modify the contained code, retrieve sub-blocks, save/load to/from files, etc. *Code Blocks* can either be created manually, loaded from files, or created automatically by the ISG or through *Code Minimization* by the framework. The latter two will be discussed in the upcoming Section 4.2 and Section 4.4.

4.2 The Grammar-based ISG

RVVTS comes with a grammar-based ISG for 32 bit (RV32) and 64 bit (RV64) RISC-V configurations. The ISG provides support for the base integer (I) and the RVV extensions, which is the focus in this work. Furthermore the generator partially supports floating-point (F/D) as far as necessary to handle RVV floating-point.

The ISG uses a context-free grammar to create syntactically valid instruction sequences. The grammar consists of *non-terminal* and *terminal symbols*. When invoked, the generator randomly selects expansion candidates for *non-terminal symbols* until only *terminal symbols* remain. The result is returned as *Code Fragment*.

Context-free grammars are highly efficient for generating simple sequences. However, the expressiveness of such grammars is too limited when it comes to more complex sequences. One example of this is the generation of bounded values, possibly even dynamically parameterized, such as the generation of an address in a specific range. More complex examples are (address-range) bounded

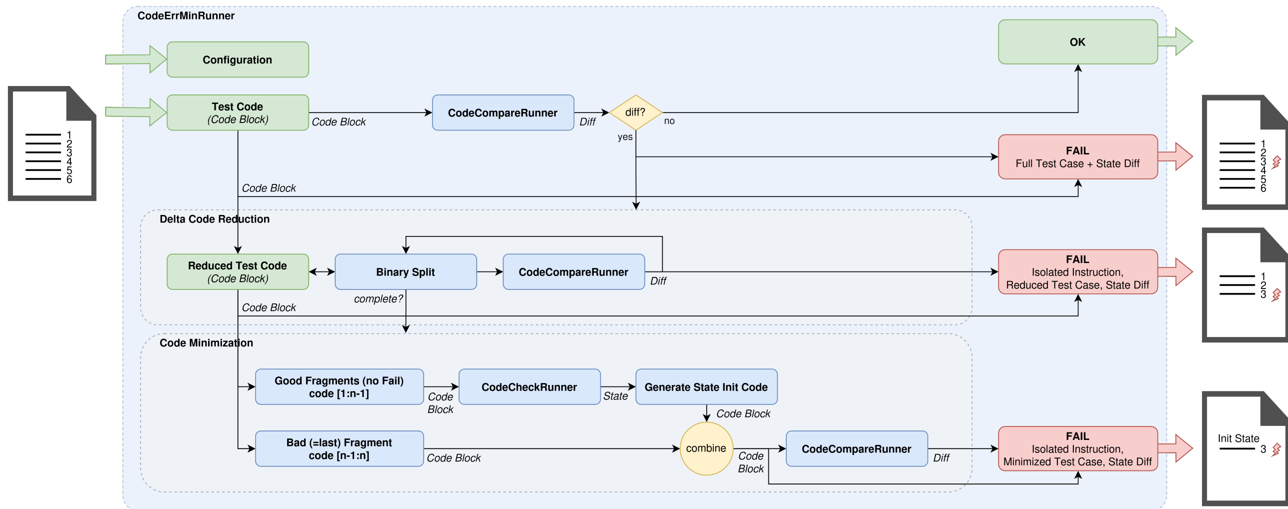


Figure 1: *CodeErrMinRunner*

load/store operations, which consist of multiple instructions dependent on each other. To handle such cases efficiently, we extend the context-free grammar with special *function symbols* associated with Python functions. The ISG expands such *function symbols* by calling the associated function, which can provide context-sensitive expressiveness. The result of the function is then integrated in the generated *Code Fragment*. In our ISG, functions are for example used to generate immediate values, register allocations, valid values for CSRs and bounded load/stores.

4.3 RVVTS's Building Blocks: *Runners*

The *RVVTS* framework is designed in modular, object oriented design paradigm with expandability in mind. Central elements of the framework are the so called *Runners*.

Basic *Runners* handle fundamental tasks as process execution, timeout handling, logging and archiving of results. Extended *Runners* add specific functionality to the framework and are created by specializations and aggregations of basic and other extended *Runners*. All *Runners* are controlled via central configuration data structure, that contains target definitions (e.g. RISC-V configuration, memory map) and other *Runner* specific settings. In the following, we provide a brief overview of the most important low- and mid-level *Runners* with regard to this paper.

BuildRunner takes a *Code Block* and a *Machine State*. The *Runner* creates a instrumented program (e.g. initialization with given *Machine State*, trap counting, dump of *Machine State* after execution) and invokes a compiler to create a *Executable and Linkable Format* (ELF) binary.

Execution Runners take a ELF binary, execute that binary on a specified target and extract the *Machine State* after the run. At the time of writing, *RVVTS* comes with *Execution Runners* to run *Spike*, which is used as a golden reference model, and *RISC-V VP++* and *QEMU* as supported DUTs. *Spike* is the official RISC-V reference simulator maintained by *RISC-V International* [13].

Support for other SW targets (e.g. simulators, emulators) or even HW targets can be introduced by adding new *Execution Runners*. Requirements for new targets are, that (i) the target can be started up in a controlled way with a given ELF binary and zeroed-out memory, (ii) the target can be stopped when reaching a specified PC position (e.g. breakpoint), and (iii) the targets architectural state can be extracted (e.g. memory dump). One way to handle these requirements is by utilizing the framework's generic *GNU Debugger* (GDB) client *Runner*. This *GDBRunner* is able to connect to any targets providing a *GDB Remote Debug Protocol* (RDB) interface and is also used by the *Execution Runners* for *RISC-V VP++* and *QEMU*.

There is also another, special form of *Execution Runner* that is used to measure the functional coverage of the given binaries. This *Execution Runner* uses the free, but closed-source *riscvOVPsim* simulator from *Imperas* [11]. The *riscvOVPsim* functional coverage metric for RVV measures the use (hits) of specified RVV instructions and their parameters. The maximum number of points of this metric consists of hits on instructions, all possible integer, floating-point and vector register addresses, and positive/negative integer register and immediate values.

RefCovRunner takes a ELF binary as input and uses the *Execution Runners* for *Spike* and *riscvOVPsim* to provide the *Machine State* and functional coverage values.

CodeCheckRunner takes a *Code Block* as input. It first uses a *BuildRunner* for instrumentation and build. The resulting ELF binary is then fed into a *RefCovRunner* which provides the *Machine State* and functional coverage values of the executed *Code Block*.

CodeCompareRunner also takes a *Code Block* as input and uses the *BuildRunner* for instrumentation and build. The resulting ELF binary is fed in a *RefCovRunner* and any other given *Execution Runner*, called DUT *Runner* in this context. The *Runner* provides a comparison of the resulting *Machine States* and also the functional coverage values of the executed *Code Block*.

4.4 Single Instruction Isolation with Code Minimization

Next, we focus on the ***CodeErrMinRunner*** which implements our novel *Single Instruction Isolation* with *Code Minimization* technique. The *Runner* is shown in Figure 1. We first consider the upper part of the figure. The *Runner* takes a *Code Block* and first uses a *CodeCompareRunner* to determine, if there are any deviations in

Algorithm 1 Delta Code Reduction: Binary Search Algorithm

```
    Input: Failing Code Block
    Output: Position of the first failing Code Fragment.
 1: good ← 0
 2: bad ← codeblock.len()
 3: test ← codeblock.len()
 4: while (bad − good) > 1 do
 5:     if CodeCompareRunner.run(codeblock.subpart(0, test)) == FAIL then
 6:         bad ← test
 7:         test ← test − ((bad − good)/2)        ▷ Reduce by half search area
 8:     else
 9:         good ← test
10:         test ← test + ((bad − good)/2)        ▷ Extend by half search area
11: return bad
```

the *Machine State*. If not, the run is considered a *pass*, the results (e.g. functional coverage values) are documented and the run is complete. In case of a *Machine State* deviation, the run is considered a *fail*. The results, including the failing *Code Block*, are documented, and the *Code Block* is given to the *Delta Code Reduction* stage.

Delta Code Reduction is seen in the mid part of Figure 1. In this reduction stage, the fail in the *Code Block* is isolated using a binary search technique presented in Algorithm 1. The algorithms keeps track of the last found *good* and *bad* code positions, and a current *test* code position. In each iteration, a sub-part of the *Code Block* starting from 0 to the *test* position is executed using a *CodeCompareRunner*. The result is checked for a fail, the code positions are updated accordingly and the search area is halved for the next iteration. The algorithm stops once it has found the earliest single bad *Code Fragment*, which is located at the *bad* position. The results of the last run and the reduced failing *Code Block*, are documented.

The *Delta Code Reduction* stage results in a reduced *Code Block* consisting of a number of non-failing *Code Fragments* and a single failing *Code Fragment* at the end. Based on this information, it is already possible to isolate a single failing instruction (the last *Code Fragment*). However, the *Code Block* may still contain many more *Code Fragments*, which makes analysis difficult. The *Code Block* is therefore given to a final stage, the *Code Minimization*, seen in the lower part of Figure 1. In this stage, the *Code Block* is split into two separate blocks: The first *Code Block* contains all, non-failing *Code Fragments* just before the failing fragment. The second *Code Block* contains only the last failing *Code Fragment*.

The first block is executed by a *CodeCheckRunner* to extract the *Machine State* after execution of the non-failing fragments. The initialization code generated from this *Machine State* is then combined with the second *Code Block* containing only the single failing fragment. The result is a much easier to analyze minimized *Code Block* containing only the *Machine State* initialization and a single failing *Code Fragment*. In a final step, this minimized *Code Block* is executed by a *CodeCompareRunner* to obtain the final results. These results and the minimized *Code Block* are then documented.

An important variant derived from the *CodeErrMinRunner* is the ***TestsetCodeErrMinRunner***, which is used to run pre-generated test sets in our case studies, in Section 5. *TestsetCodeErrMinRunner* executes a series of *Code Blocks* provided as the files and collects all obtained results. The *Runner* takes a path and a file matching pattern as input. Files matching the given pattern are loaded as *Code Block* and executed as described above. The results of all fails, including the reduced and minimized *Code Blocks* are documented for later analysis.

4.5 Coverage Guided Test Set Generation

With the ***CovGuidedTestsetGenerator***, the *RVVTS* framework provides a *Runner* able to generate dense *Code Blocks* with high functional coverage. Multiple, independent instances of these *Runners* can be used in parallel to generate large sets of such *Code Blocks*. Such sets are particularly useful for testing, as we will demonstrate in our case studies in Section 5.

The generator uses the grammar-based ISG from Section 4.2 and the *CodeCheckRunner*, presented in Section 4.3. The generation process starts with an empty *Code Block* with zero functional coverage and consists of two alternating phases: (i) The *Extension* phase, in which the *Runner* tries to extend the *Code Block* with new *Code Fragments* to improve coverage, and (ii) and the *Reduction* phase, in which the *Runner* tries to densify the *Code Block* by removing *Code Fragments* without hurting coverage. Switching between these phases is determined by a configurable number of iterations for each phase. The process can be stopped at any iteration by the program that instantiated the *Runner*. Typical examples of this are stopping after a certain number of iterations or time, or after reaching a certain coverage value.

We will now look at the two phases in more detail. At each iteration of the *Extension* phase, the ISG generates a new *Code Fragment*, which is inserted at a random position in a copy of the current *Code Block*. This new *Code Block* is then instrumented, compiled and executed by a *CodeCheckRunner*, which returns a *Machine State* and the coverage value. These results must meet two requirements: (i) The *Machine State* must be valid, which will be discussed below in more detail, and (ii) the coverage obtained for the new *Code Block* must be maintained or improved compared to the original *Code Block*. If one of these requirements is not met, the new *Code Block* is rejected and the original *Code Block* is used in the next iteration. If all requirements are met, the new *Code Block* is used for for the next iteration.

At each iteration of the *Reduction* phase, a random *Code Fragment* is removed from a copy of the current *Code Block*. Also in this phase, the new *Code Block* is executed via a *CodeCheckRunner*. The results are checked for similar requirements as above (validity and coverage), with one important difference: The coverage obtained for the new *Code Block* must stay the same wrt. the original *Code Block*. Again the new *Code Block* is only used for the next iteration, if these requirements are met.

The validity property mentioned above consists of at least the minimum requirements that the generated *Code Block* must be compilable and executable. However, additional requirements are configurable. At the time of writing, the only configurable property is a Boolean switch that controls whether the resulting *Code Block* is allowed to cause traps in the execution. As we will see in our case studies below, in Section 5, such a switch is crucial for generating test sets targeted for pure positive, and positive/negative testing.

5 CASE STUDIES

In this section, we demonstrate the effectiveness of *RVVTS* and the novel *Single Instruction Isolation* with *Code Minimization* technique. First, in Section 5.1, we present four test sets generated with *CovGuidedTestsetGenerator* (Section 4.5), that can be used in the verification process of RV32 and RV64 based RVV implementations. After this, we use the *TestsetCodeErrMinRunner* (Section 4.4)

Table 1: Test Sets pre-generated with *CovGuidedTestsetGenerator*

Test set	RISC-V Config	Million Code Fragments	Million RVV Instr.	Functional Coverage (*riscvOVPsim* RVV) Points	Percent
Valid Sequences (VS)	RV32	1.78	1.25	30,500 / 31,894	95.63
	RV64	1.69	1.21	31,410 / 33,076	94.96
Invalid+Valid Sequences (IVS)	RV32	1.70	1.31	30,919 / 31,894	96.94
	RV64	1.64	1.26	31,952 / 33,076	96.60
Merged Sequences (MS)	RV32	3.49	2.56	30,920 / 31,894	96.95
(VS + IVS)	RV64	3.33	2.47	31,952 / 33,076	96.60

including *Single Instruction Isolation* with *Code Minimization* to apply the pre-generated test sets on two DUTs. The first DUT, *RISC-V VP++* [36] is considered in Section 5.2. The second DUT, *QEMU* [6], is covered in Section 5.3. For both DUTs, we perform a result analysis and discuss previously undetected bugs. All newly found bugs are reported to the respective open-source projects.

5.1 Pre-Generated Testsets

We use the *CovGuidedTestsetGenerator*, presented in Section 4.5 to generate four test sets by varying two configuration settings: (i) the RISC-V configuration (RV32, RV64), and (ii) the Boolean switch, that controls whether the generated test sets are allowed to contain instruction sequences that trigger traps when executed.

The former, the RISC-V configuration is varied to enable support for testing RV32 and RV64 DUTs and thus to extend the usability of the pre-generated test sets. The latter switch enables support for different test strategies, namely having separate test sets for pure positive testing, containing only *Valid Sequences* (VS) and positive/negative testing, containing *Invalid+Valid Sequences* (IVS).

For the generation of all test sets, the following global configuration is chosen: (i) At least 3 MiB read/writeable memory located at address $0x80020000$ is expected. The memory is partitioned in a $1,536$ KiB code, a 10 KiB dump (*Machine State*) and a $1,526$ KiB data area. Generated bounded loads can access the full memory area, while generated bounded stores can only access the data area. (ii) Code is generated for RVV implementations with a `VLEN` of up to 512 bits and a `ELEN` of 64 bits.

Table 1 summarizes the major characteristics of (i) the four generated test sets with *Valid Sequences* (VS) and with *Invalid+Valid Sequences* (IVS) for RV32 and RV64, respectively, and (ii) the merged test sets with *Merged Sequences* (MS) (i.e. VS + IVS), for RV32 and RV64, respectively. For each test set, the table shows the number of generated code fragments, the number of generated RVV instructions, and the achieved RVV functional coverage. As can be seen in Table 1, all test sets reach a functional coverage above 94%.

The pre-generated test sets can be used to test RV32 and RV64 based RVV implementations and are available as open-source on GitHub.

5.2 *RISC-V VP++*

In this case study, we consider the recently released *RISC-V VP++* with support for RVV 1.0. We use the *TestsetCodeErrMinRunner* to apply the test sets generated in Section 5.1 on different variants and versions of the VP.

Table 2 shows the result of the application of the RV32 and RV64 VS, IVS and MS test sets on *RISC-V VP++*. We obtained results for two versions of the VP: (i) The *initial* version of *RISC-V VP++* with support for RVV, presented in [36]. (ii) The *latest* version of *RISC-V VP++* at the time of writing (git commit hash *c354bfea*). Due to a fundamental bug related to the integration of RVV in the *initial* version of the RV32 VP variant, the results of the test runs are not valid for this case and are therefore listed as N/A in the table.

Table 2: Test Sets executed on *RISC-V VP++*

Test set	RISC-V Config	VP Version	Detected Fails	Isolated Failing Instructions
VS	RV32	*initial*	N/A	N/A
		latest	1,656	9
	RV64	*initial*	39,842	409
		latest	1,301	9
IVS	RV32	*initial*	N/A	N/A
		latest	195	9
	RV64	*initial*	112,015	539
		latest	183	9
MS	RV32	*initial*	N/A	N/A
		latest	1,849	10
	RV64	*initial*	151,857	542
		latest	1,484	9

For each run, Table 2 shows the overall number of detected fails and the number of isolated, unique RVV instructions causing these these fails. As explained in Section 4.4, a fail is defined as deviating *Machine State* extracted after execution on a reference simulator (*Spike*) and the DUT.

First, we focus on the differences from the *initial* to the *latest* RV64 VP version on a high level. It can be seen, that the number of detected deviations and identified instructions drops significantly from the *initial* to *latest* VP version. This is not surprising due to the fact that the *latest* version contains 42 RVV related bug fixes compared to the *initial* version. These 42 fixes can be roughly classified in three categories: (i) 6 fixes are related to the integration of RVV in the RV32 and RV64 ISSs (e.g. access to RVV CSRs), (ii) 11 fixes are related to the function of specific RVV instructions or instruction groups (e.g. incorrect result for vector remainder, ...), and (iii) 25 fixes are related to handling illegal parameter combinations or configurations for RVV instructions (e.g. invalid type settings, use of invalid registers, ...).

Bugs in the first category usually have a major impact and their effects can be seen quickly in tests. For example, a generic bug in floating-point handling can affect the behaviour of many floating-point instructions. However, bugs in this category are often hard to track down due to their generic effects. With respect to our results in Table 2, such kind of bugs can be detected equally well with the VS and IVS test sets.

Bugs in the second category are related to the function of specific RVV instructions or instruction groups. For example, incorrect results of a RVV remainder instruction for certain parameter values.

Bugs of this kind can also be detected with the VS and IVS test sets. However, since the VS test set is likely to contain more valid parameter combinations for RVV instructions than a IVS test set of roughly the same size, the VS test set is more suited to detect such kind of bugs. This can also be seen in Table 2 for the *initial* VP. For the VS and IVS test sets, we can identify 409 and 539 unique instructions causing fails, respectively. However, for the merged MS test set we can identify 542 unique instructions causing fails. This means, that $542 - 539 = 3$ instructions were identified only with the VS test set containing valid sequences.

Bugs in the third and last category are related to illegal parameter combinations or configurations. The functionality of RVV instructions is heavily dependent on the architectural state. Two prominent examples for this are *dynamic typing* and *register grouping*. An example of the former, floating-point operations on 8 bit elements, was already presented in Section 1. The latter, *register grouping*, is a special feature of RVV to group the 32 available vector registers together to either 16, 8 or 4 registers with 2, 4 or 8 times the size of a single register. For example, if *register grouping* is not used, the instruction `vadd v0, v1, v2` is valid. If *register grouping* is set to 2, only even numbered registers are allowed. In this case, the exact same instruction is invalid and will cause a trap.

Instruction sequences containing illegal parameter combinations or configurations are by principle not included in the VS test set. Bugs related to invalid sequences can only be detected with negative testing and hence with the IVS test set. This can also be seen in Table 2: By subtracting the number of instructions with fails detected by the VS test set (409) from the number detected by the merged MS test set (542), we get $542 - 409 = 133$ instructions with fails detected only by the IVS test set.

We now focus on the results of the *latest* version of *RISC-V VP++* in Table 2. Comparing the absolute number of fails detected by the VS test set ($1,656$ and $1,301$), with the IVS test set (195 and 183), shows a large difference. The much lower values for the IVS test set indicates, that most remaining problems in *RISC-V VP++* are related to the first and second category described above. Overall, for the merged MS test set, we can observe 10 and 9 isolated instructions with fails for RV32 and RV64, respectively. By analyzing the automatically generated minimized test cases and the difference of the extracted *Machine States* after the run, we can identify four types of potential bugs: (i) Deviations for floating-point reduction operations, more specifically the unordered floating-point sums `vfredusum.vs` and `vfwredusum.vs`, (ii) Unexpected traps of floating-point to/from integer conversion operations, (iii) Unexpected traps and invalid results of `vfrsqrt7.v`, and (iv) Crashes of the VP related to `vrem.vv`. We will now analyze the first two and most interesting types in detail.

The first type can be identified as false-positive fail of the instructions `vfredusum.vs` and `vfwredusum.vs`, which are specified to provide a sum of floating-point vector elements. It is important to note that floating-point calculations are not associative. Consequently, the order in which elements are processed matters and can lead to different results. To avoid any potential ambiguities, RVV specifies two variants of these sum operations: (i) the *ordered* variants, `vfredosum.vs` and `vfwredosum.vs`, which sums the values strictly in element order (0, 1, 2, ...), and (ii) the *unordered* variants, `vfredusum.vs` and `vfwredusum.vs`, which offer more freedom regarding the order of processing elements. While the former brings the advantage of comparability and reproducibility, the latter may offer advantages in terms of performance. However, RVV also specifies, that *ordered* implementations are also valid *unordered* implementations. This fact is exploited by the developers of *RISC-V VP++*, where `vfredusum.vs` and `vfwredusum.vs` use the exact same implementations as `vfredosum.vs` and `vfwredosum.vs`, respectively. As this is not the case for the reference simulator *Spike*, some results of the operations differ and are identified as fails by *RVVTS*. However, since we have the minimized test cases and *Machine States*, it is easy to verify, that the *unordered* VP results match the *ordered* reference results. We semi-automatically replace the *unordered* variants with the *ordered* variants in all generated minimized test cases and perform a re-run on the reference simulator. Comparing the resulting *Machine States* with the states of the previous run on the VP shows no differences and therefore, all detected fails of `vfredusum.vs` and `vfwredusum.vs` are false-positives.

The second type of potential bugs concerns conversion instructions of floating-point to/from integer with widening and narrowing result width (e.g. `vfwcvt.f.x.v`, `vfncvt.x.f.w`, ...). A quick analysis of the results of the corresponding minimized test cases shows, that in all cases the VP has unexpected traps, while the instructions are successfully executed on the reference simulator. Since the VP can provide detailed output on causes of RVV traps, and *RVVTS* automatically records all output of failed runs, we can easily track down the cause of all traps. Analyzing the output shows that the cause of the traps is the same in all detected failures, namely that half-precision floating-point is not supported. Since half-precision floating-point is not used in any of the failing test cases this must be a bug in *RISC-V VP++*. A code analysis of the RVV implementation in *RISC-V VP++* reveals that there is indeed a faulty assertion related to detecting the use of the not-yet-supported half-precision floating-point.

Table 3: Test Sets executed on *QEMU* (git hash 02e16ab9f4)

Test set	RISC-V Config	Detected Fails	Isolated Failing Instructions
VS	RV32	10,575	114
	RV64	6,548	112
IVS	RV32	8,667	167
	RV64	8,463	166
MS	RV32	19,242	168
	RV64	15,011	166

5.3 *QEMU*

In this case study, we consider *QEMU* in its latest version at the time of writing, *v9.0.0-rc3*. First, we use the *TestsetCodeErrMinRunner* to apply the test sets generated in Section 5.1 on RV32 and RV64 variants of the emulator. Then, we perform a result analysis leveraging *RVVTS* and *Single Instruction Isolation* with *Code Minimization*. Finally, we discuss a selection of interesting newly discovered bugs in *QEMU*.

Table 3 shows the results of the application of the RV32 and RV64 VS, IVS and MS test stets on *QEMU*. For each run, the table shows the overall number of detected fails and the number of isolated, unique RVV instructions causing these these fails.

First, we focus on the difference between the VS (positive testing) and IVS (positive/negative testing) test sets. Subtracting the number of failing instructions detected by the IVS test sets (167 and 166) from the corresponding numbers of the merged MS test sets (168 and 166), gives us 1 and 0 instructions for RV32 and RV64, which are detected only by the VS test sets with valid sequences. Similarly, subtracting the number failing instructions of the VS test sets (114 and 112) from the MS test sets (168 and 166), gives us 54 instructions for RV32 and RV64, which are only detected by the IVS test sets with invalid sequences. We can see that in the case of *QEMU*, the IVS test sets, and therefore the negative tests, are very important.

We now focus on the failing instructions and perform a exemplary analysis of the results and a *cause analysis leveraging RVVTS and Jupyter notebooks*. Please note that the workflow presented below is only one of many possible workflows supported by *RVVTS*.

All failed instructions are classified according to their function (e.g. *load, store*), or a main characteristic (e.g. *float widening operation*). For each class, all generated minimized cases form a new test set which is loaded and executed by *RVVTS* until the first fail. By studying the failing minimized test case and the resulting *Machine State* difference, assumptions of the cause and corresponding measures are derived. These measures may include, for example, experimental fixes in the DUT, and semi-automatically applied adaption of test cases or result checks. After documenting the assumed cause and implementing corresponding measures, the automated execution of all minimized test cases is repeated. Depending on whether the assumptions and measures are correct and whether all fails are due to the same cause, none, some, or all cases may still fail. These still failing test cases are again automatically documented by *RVVTS* and form a new (sub-)test set. The whole process is repeated until all causes are found.

Table 4 presents the result of this process for our current *QEMU* case study. The left-most column of the table shows the formed instruction classes. For each class the number of corresponding isolated failing instructions and the number of minimized test cases for RV32 and RV64 is given. The right-most column of the table represents the results of the analysis and contains a list of brief descriptions of the isolated causes. In order to conclude our case study on *QEMU*, we now discuss two interesting causes that have been identified as previously undetected bugs in *QEMU*.

The first bug is related to the `vslide1*.vx` class of instructions, `vslide1up.vx` and `vslide1down.vx`, which only fail on RV32. Both instructions slide the elements in a vector in the given direction by one element and insert a new vector element from a given scalar integer register. On the affected RV32, the scalar integer registers are 32 bit wide. All detected failing cases have the following characteristics: (i) vector elements are configured to 64 bit, and (ii) the most significant bit in the 32 bit scalar integer register is set, which can be interpreted as negative value. The corresponding results show, that the reference simulator *Spike* and *QEMU* provide the same result on the lower 32 bits of the newly introduced vector element, which also matches the value of the scalar integer register. However, the higher 32 bit of the value differ: While these bits are all set (= 1) by the reference simulator, corresponding to a sign extension of the scalar value, they are all cleared (= 0) by *QEMU*. This is definitely a bug in *QEMU*, as the RVV specification version 1.0 clearly states that a sign-extend must

Table 4: Result Analysis for QEMU

RVV Instruction Class	**#Isolated Instructions** (RV32/RV64)	**#Minimized Test Cases** (RV32/RV64)	**Isolated Causes**
Load	80 / 78	4,492 / 3,285	RV32: 64 bit indices not supported Traps on first 4KiB of memory Fractional lmul in QEMU allowed
Store	46 / 48	3,149 / 727	RV32: 64 bit indices not supported Fractional lmul in QEMU allowed
Reduction	16 / 16	7,095 / 6,714	Invalid behavior with vl=0
Float widening	21 / 21	843 / 772	Invalid support for half-precision
`vset*`	3 / 3	3,409 / 3,488	Extended support for fractional lmul
`vslide1*.vx`	2 / 0	226 / 0	RV32: Broken sign-extend of 32 to 64 bits

be performed in these cases. Interestingly, the very similar vector integer move instructions, which copy a scalar value from an integer register (`vmv.v.x`) or an immediate (`vmv.v.i`) into a vector, behave as expected for negative values.

The second bug discussed here concerns all RVV *reduction* instructions. As the name suggests, all these instructions reduce a vector to a single element. The `vredsum.vs vd, vs2, vs1` instruction is an example for this. The instruction sums all elements in vector register `vs2` and the first element of vector register `vs1`. The result is written to the first element of vector register `vd`. All detected failing cases share one commonality: The vector length (`vl`) is set to zero. The corresponding results show, that the reference simulator *Spike* does not perform a write to `vd` in these cases, which is the correct behavior defined in the RVV specification. In contrast, *QEMU* writes the value of the first element of `vs2` to `vd`, which contradicts the specification and is therefore a clear bug in *v9.0.0-rc3* of *QEMU*. A comparison run on an earlier version, *v8.0.4*, of *QEMU* also shows that this is a new bug introduced somewhere in between *v8.0.4* and *v9.0.0-rc3* of *QEMU*.

6 CONCLUSIONS

In this paper, we have introduced the modular and open-source *RVVTS* framework which comes with our novel *Single Instruction Isolation* technique for positive and negative testing of RVV implementations, addressing the challenges posed by the complexity of the 600+ highly configuration-dependent RVV instructions.

The effectiveness of *RVVTS* was demonstrated in our case studies, where we (i) generated test sets with functional coverage >94% for positive and negative testing of 32 and 64 bit RVV implementations, (ii) applied the test set on the *RISC-V VP++* and *QEMU* resulting in minimized test cases and isolated failing instructions using our *Single Instruction Isolation* with *Code Minimization* technique, and (iii) traced down the causes of the failed isolated instructions to find the root cause.

Overall, we confirmed 3 new bugs in the *RISC-V VP++* and 2 new bugs in *QEMU* (and 7 more are to be analyzed where we will consider the formal RISC-V Sail specification model to finally clarify potential ambiguities in the RVV specification). The generated test sets and the *RVVTS* framework are available as open-source on GitHub. Moreover, our findings are reported to the respective open-source projects.

ACKNOWLEDGMENTS

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

REFERENCES

- [1] 2021. RISC-V Compliance Task Group. https://github.com/riscv/riscv-compliance.
- [2] 2021. RISC-V Torture Test Generator. https://github.com/ucb-bar/riscv-torture.
- [3] 2022. RISC-V V vector extension. https://github.com/riscv/riscv-v-spec.
- [4] 2023. IEEE Standard for Standard SystemC Language Reference Manual. https://doi.org/10.1109/IEEESTD.2023.10246125
- [5] 2024. FORCE-RISCV RISC-V Instruction Sequence Generator (ISG). https://github.com/openhwgroup/force-riscv.
- [6] 2024. QEMU A generic and open source machine emulator and virtualizer. https://www.qemu.org.
- [7] 2024. RISC-V Architecture Test SIG. https://github.com/riscv-non-isa/riscv-arch-test.
- [8] 2024. RISC-V Formal Verification Framework. https://github.com/YosysHQ/riscv-formal.
- [9] 2024. RISCV-DV. https://github.com/google/riscv-dv.
- [10] 2024. RISCV Sail Model. https://github.com/rems-project/sail-riscv.
- [11] 2024. riscvOVPsim Imperas RISC-V Instruction Set Simulator (ISS). https://www.imperas.com/riscvovpsim-free-imperas-risc-v-instruction-set-simulator.
- [12] 2024. Siemens EDA Questa Formal Verification Apps. https://eda.sw.siemens.com/en-US/ic/questa/formal-verification.
- [13] 2024. Spike RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim.
- [14] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. 2004. Genesys-Pro: innovations in test program generation for functional processor verification. *IEEE Design & Test of Comp.* (2004), 84–93.
- [15] Niklas Bruns, Daniel Große, and Rolf Drechsler. 2020. Early Verification of ISA Extension Specifications Using Deep Reinforcement Learning. In *ACM Great Lakes Symposium on VLSI*. 297–302.
- [16] Brian Campbell and Ian Stark. 2014. Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation. In *Formal Methods for Industrial Critical Systems*. 185–199.
- [17] Mikhail Chupilko, Alexander Kamkin, Artem Kotsynyak, and Andrei Tatarnikov. 2017. MicroTESK: Specification-Based Tool for Constructing Test Program Generators. In *Haifa Verification Conference*. 217–220.
- [18] Keerthikumara Devarajegowda, Mohammad Rahmani Fadiheh, Eshan Singh, Clark W. Barrett, Subhasish Mitra, Wolfgang Ecker, Dominik Stoffel, and Wolfgang Kunz. 2020. Gap-free Processor Verification by S2QED and Property Generation. In *Design, Automation and Test in Europe*. 526–531.
- [19] Roger Espasa, Mateo Valero, and James E. Smith. 1998. Vector Architectures: Past, Present and Future. In *International Conference on Supercomputing*. 425–432.
- [20] Shai Fine and Avi Ziv. 2003. Coverage directed test generation for functional verification using Bayesian networks. In *Design Automation Conf.* 286–291.
- [21] M.J. Flynn. 1966. Very high-speed computing systems. *IEEE* 54, 12 (1966), 1901–1909.
- [22] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side. In *Design Automation Conf.* 1–6.
- [23] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer.
- [24] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Towards Specification and Testing of RISC-V ISA Compliance. In *Design, Automation and Test in Europe*. 995–998.
- [25] Vladimir Herdt, Daniel Große, Eyck Jentzsch, and Rolf Drechsler. 2020. Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study. In *Forum on Specification and Design Languages*. 1–7.
- [26] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing. In *Design, Automation and Test in Europe*. 360–365.
- [27] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *Symposium on Security and Privacy*. 1286–1303.
- [28] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. 2011. Feedback-Based Coverage Directed Test Generation: An Industrial Evaluation. In *Haifa Verification Conference*. 112–128.
- [29] Victor Jimenez, Mario Rodriguez, Marc Dominguez, Josep Sans, Ivan Diaz, Luca Valente, Vito Luca Guglielmi, Josue V. Quiroga, R. Ignacio Genovese, Nehir Sonmez, Oscar Palomar, and Miquel Moreto. 2023. Functional Verification of a RISC-V Vector Accelerator. *IEEE Design & Test of Comp.* 40, 3 (2023), 36–44.
- [30] Yoav Katz, Michal Rimon, and Avi Ziv. 2012. Generating instruction streams using abstract CSP. In *Design, Automation and Test in Europe*. 15–20.
- [31] Lucas Klemmer and Daniel Große. 2021. EPEX: Processor Verification by Equivalent Program Execution. In *ACM Great Lakes Symposium on VLSI*. 33–38.
- [32] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *International Symposium on Software Testing and Analysis*. 261–272.
- [33] Manfred Schlägl and Daniel Große. 2023. GUI-VP Kit: A RISC-V VP Meets Linux Graphics - Enabling Interactive Graphical Application Development. In *ACM Great Lakes Symposium on VLSI*. 599–605.
- [34] Manfred Schlägl and Daniel Große. 2024. Bounded Load/Stores in Grammar-based Code Generation for Testing the RISC-V Vector Extension. In *RISC-V Summit Europe*.
- [35] Manfred Schlägl, Christoph Hazott, and Daniel Große. 2024. RISC-V VP++: Next Generation Open-Source Virtual Prototype. In *Workshop on Open-Source Design Automation*.
- [36] Manfred Schlägl, Moritz Stockinger, and Daniel Große. 2024. A RISC-V "V" VP: Unlocking Vector Processing for Evaluation at the System Level. In *Design, Automation and Test in Europe*. 1–6.
- [37] Eshan Singh, David Lin, Clark W. Barrett, and Subhasish Mitra. 2018. Logic Bug Detection and Localization Using Symbolic Quick Error Detection. *IEEE Transactions on Computer Aided Design of Circuits and Systems* (2018).
- [38] Eshan Singh, Florian Lonsing, Saranyu Chattopadhyay, Maxwell Strange, Peng Wei, Xiaofan Zhang, Yuan Zhou, Deming Chen, Jason Cong, Priyanka Raina, Zhiru Zhang, Clark W. Barrett, and Subhasish Mitra. 2020. A-QED Verification of Hardware Accelerators. In *Design Automation Conf.* 1–6.
- [39] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and UC Berkeley.
- [40] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. SiFive Inc. and UC Berkeley.