RVVTS: A Modular, Open-Source Framework for Positive and Negative Testing of the RISC-V "V" Vector Extension (RVV)

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

Abstract

In this extended abstract, we summarize the work from [1], where we presented the modular, open-source framework *RVVTS* for positive and negative testing of the *RISC-V "V" Vector Extension* (RVV) with its 600+ highly configurable instructions. The framework comes with a grammar-based, coverage-guided *Instruction Sequence Generator* (ISG), automation for instrumentation, build and run of test-cases, and the ability to detect differences in architectural states after runs. At the heart of the framework is our novel *Single Instruction Isolation* with *Code Minimization* technique which allows to reduce manual result analysis of failing test cases significantly. By applying *RVVTS* to the *RISC-V VP++ Virtual Prototype* and the *QEMU* emulator, we confirmed 3 new bugs in the *RISC-V VP++* and 2 in *QEMU* (and 7 more are to be analyzed). *RVVTS*, as well as the pre-generated test sets are available as open-source on GitHub[*].

1 Extended Abstract

RISC-V [2, 3], 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 [4]. 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 ISGs, to facilitate exhaustive verification processes.

Let us now specially look on the complexity challenge introduced by the RVV with its 600+ instructions. RVV brings extensive *Single Instruction, Multiple Data* (SIMD) capabilities to RISC-V, enabling it 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 [5]. 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* [6]. 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* [7]. 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

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

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.

In this work, 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 Instruction 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.

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 very efficiently. 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. 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. 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. In our ISG, functions are for example used to generate immediate values, register allocations, valid values for CSRs and bounded load/stores.

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*. 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. Simple examples for *Runners* are the *BuildRunner*, which handles instrumentation and build, and the *Execution Runners*, which handle execution on target platforms. A more complex example is the aggregated *CodeCompareRunner*, which runs a program on a reference simulator (*Spike* [8]) and a DUT, and provides a comparison of the resulting architectural states and also functional coverage values.

When the framework detects a difference in the architectural state after the run of a test case with *CodeCompareRunner*, the framework considers this a potential fail. As discussed above, investigating many such potential fails manually is very labor intensive. *RVVTS* solves this problem by using our novel *Single Instruction Isolation with Code Minimisation* technique, which is divided into two phases. First, the potential failing test-case is fed into *Delta Code Reduction*, where a binary search technique is used to identify the first instruction causing a deviation in architectural state. The resulting *reduced* test-case contains a cutout of the original test-case ranging from the first instruction, up to and including the identified potentially failing instruction. This reduced test-case is fed into the second phase, the *Code Minimization*. Here, the test-code is split before the last (potentially failing) instruction. The first part contains only instructions that cause no differences in architectural state. The second part contains only the potential failing instruction identified before. The first part is run on a reference simulator and the resulting architectural state is extracted. This state is then converted to initialization code, which brings a machine to exactly the extracted state when executed. Finally, the initialization code is combined with the second part (single, potentially failing instruction). The resulting *minimized* test-case contains only the initialization code and a single potentially failing instruction, and is therefore much easier to inspect.

In our case studies, we demonstrate the effectiveness of

RVVTS and the novel *Single Instruction Isolation* with *Code Minimization* technique. We use use the framework to generate ready-to-use test sets for positive and negative testing of RVV implementations with a `VLEN` of 512 bits for RV32 and RV64, respectively. In total we have four test sets: For both RV32 and RV64 we have one containing only *Valid Sequences* (VS) for pure positive testing and one containing *Invalid+Valid Sequences* (IVS) for positive/negative testing. The merged test set with *Merged Sequences* (MS) (i.e. VS + IVS) for RV32 contains 2.56 million RVV instruction with a **functional coverage of 96.95%**. The MS test set for RV64 contains 2.47 million RVV instruction with a **functional coverage of 96.60%**.

The tests sets are applied on two DUTs implementing RVV for RV32 and RV64 in its ratified version 1.0, namely the open-source SystemC [9, 10] TLM based *RISC-V VP++* [11, 12, 13] *Virtual Prototype* (VP) and the open-source *QEMU* emulator [14]. The MS test set on *RISC-V VP++* detects 1,849 fails for RV32 and 1,484 fails for RV64. Applying *Single Instruction Isolation* with *Code Minimization* leads to significant reduction to 10 and 9 potential failing instructions, respectively. A systematic analysis of the minimized test cases confirms **3 new bugs** in *RISC-V VP++*. On *QEMU*, the MS test set detects 19,242 fails for RV32 and 15,011 fails for RV64. Applying *Single Instruction Isolation* with *Code Minimization* isolates 168 and 166 potential failing instructions, respectively. Here, the systematic analysis confirms **2 new bugs** and 7 additional potential bugs in *QEMU* that need to be investigated further.

For future work, we plan to analyze the 7 potential bugs in *QEMU*, where we will consider the formal RISC-V Sail specification model [15] to finally clarify potential ambiguities in the RVV specification. Furthermore, we plan to demonstrate the application of *RVVTS* and the pre-generated test sets on RTL models and on real hardware.

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.

2 Literature

[1] M. Schlägl and D. Große, “Single instruction isolation for RISC-V vector test failures,” in *International Conference on Computer-Aided Design*, 2024.

[2] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and UC Berkeley, 2019.

[3] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and UC Berkeley, 2019.

[4] “RISC-V compliance task group,” https://github.com/riscv/riscv-compliance, 2021.

[5] “RISCV-DV,” https://github.com/google/riscv-dv, 2024.

[6] “FORCE-RISCV RISC-V instruction sequence generator (isg),” https://github.com/openhwgroup/force-riscv, 2024.

[7] V. Herdt, D. Große, and R. Drechsler, “Closing the RISC-V compliance gap: Looking from the negative testing side,” in *Design Automation Conf.*, 2020, pp. 1–6.

[8] “Spike RISC-V ISA simulator,” https://github.com/riscv/riscv-isa-sim, 2024.

[9] “IEEE standard for standard SystemC language reference manual,” 2023. [Online]. Available: https://doi.org/10.1109/IEEESTD.2023.10246125

[10] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[11] M. Schlägl and D. Große, “GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development,” in *ACM Great Lakes Symposium on VLSI*, 2023, pp. 599–605.

[12] M. Schlägl, M. Stockinger, and D. Große, “A RISC-V “V” VP: Unlocking vector processing for evaluation at the system level,” in *Design, Automation and Test in Europe*, 2024, pp. 1–6.

[13] M. Schlägl, C. Hazott, and D. Große, “RISC-V VP++: Next generation open-source virtual prototype,” in *Workshop on Open-Source Design Automation*, 2024.

[14] “QEMU a generic and open source machine emulator and virtualizer,” https://www.qemu.org, 2024.

[15] “Riscv sail model,” https://github.com/rems-project/sail-riscv, 2024.