

# From Generation to Failure Categorization: An Open-Source automated RTL Verification Framework for RVV

Manfred Schlägl  
Johannes Kepler University  
Linz, Austria  
manfred.schlaegl@jku.at

Jonas Reichhardt  
Johannes Kepler University  
Linz, Austria  
jonas.reichhardt@jku.at

Daniel Grosse  
Johannes Kepler University Linz &  
DFKI Bremen  
Linz, Austria  
daniel.grosse@jku.at

## Abstract

We present a highly automated, open-source RTL verification framework for the *RISC-V Vector Extension* (RVV) 1.0 that extends the open-source *RVVTS* framework by adding RTL support and a novel *Automated Failure Categorization* (AFC) stage for scalable result analysis. Using the most recent RTL of the silicon-proven *Ara* vector processor as DUT, we generate RVV test sets that encompass both positive and negative testing, and achieve over 96% functional coverage – significantly exceeding our evaluated baseline, which achieves only 11.04%. Our framework detects more than 82k deviations, automatically minimizes about 97% of them, and clusters observed failures into 16 distinct categories.

## CCS Concepts

• **Hardware** → **Equivalence checking; Simulation and emulation**; • **Computer systems organization** → **Multiple instruction, multiple data; Reduced instruction set computing**.

## Keywords

RISC-V, RISC-V Vector Extension (RVV), RTL Verification, Processor Verification

### ACM Reference Format:

Manfred Schlägl, Jonas Reichhardt, and Daniel Grosse. 2026. From Generation to Failure Categorization: An Open-Source automated RTL Verification Framework for RVV. In *Great Lakes Symposium on VLSI 2026 (GLSVLSI '26)*, June 22–24, 2026, Canandaigua, NY, USA. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3787109.3815255>

## 1 Introduction

Exploiting *Data-Level Parallelism* (DLP) can significantly accelerate algorithms in domains such as modern machine learning and multimedia. A common approach to leverage DLP is through *Single Instruction, Multiple Data* (SIMD) extensions, which apply the same operation to multiple data elements simultaneously, i.e., a **vector**. Classical SIMD architectures, such as Intel SSE, AVX, or ARM Neon, use fixed-size vector registers, which limits flexibility and requires recompilation when the vector length changes. In contrast, vector architectures such as *RISC-V Vector Extension* (RVV) generalize

SIMD by supporting *dynamic* vector lengths (*vl*) and types (*vtype*), configured at runtime with dedicated instructions.

Among RVV implementations, *Ara* stands out as the first fully open-source processor compliant with the frozen RVV 1.0 specification (see e.g. [27, 29]). In the most recent version, it delivers very good performance on compute-intensive workloads, sustaining high utilization and scaling efficiently with core and lane counts. Moreover, *Ara* achieves state-of-the-art energy efficiency (37.8 DP-GFLOPS/W at 0.8 V; 1.35 GHz in 22 nm) [27].

However, the very features that make *Ara* compelling – RVV 1.0’s dynamic vector lengths and data types, flexible register grouping, masking, and scalable lane-based microarchitectures – substantially enlarge the state space, rendering ad hoc testbenches and simple instruction generation insufficient. We substantiate this argument with the following considerations: For a scalar RISC-V instruction (e.g. classical integer add), the architectural behavior is essentially configuration-*independent* (integer operation) or varies only by the floating-point rounding mode ( $\leq 5$  possibilities via the respective *frm Control and Status Register* (CSR)). Hence, the per-instruction configuration space is constant. In contrast, the behavior of an RVV instruction is parameterized by the dynamic CSRs *vtype* (SEW, LMUL, VTA, VMA), *vl* and *vstart*, and optional predication (e.g. masking). For fixed (SEW, LMUL), the RVV specification permits  $vl \in \{0, \dots, VLMAX\}$  and  $vstart \in \{0, \dots, vl\}$  with  $VLMAX = \lfloor LMUL \times VLEN / SEW \rfloor$ , yielding the triangular sum  $\sum_{vl=0}^{VLMAX} (vl + 1) = ((VLMAX + 1) \times (VLMAX + 2)) / 2$  distinct (*vl*, *vstart*) pairs – so at least a quadratic growth in configurations per instruction even before accounting for masks or other mode bits.<sup>1</sup> Thus, the **per-instruction configuration space for RVV is orders of magnitude larger than that in the scalar case**.

Recently, the open-source framework *RVVTS* [30] has been proposed to address this challenge. It combines coverage-guided test generation with a single-instruction isolation technique to (a) generate high-quality RVV tests and (b) automatically isolate failing instructions and provide minimized test cases, thereby drastically reducing the manual analysis effort. The *RVVTS*-generated RVV test sets achieve a functional coverage of >90% and employ both *positive and negative testing*, i.e., it generates tests that are expected to pass, verifying compliant behavior, as well as tests expected to fail, ensuring correct handling of illegal or exceptional behavior [21].

However, in general two major limitations remain: (i) the verification capabilities of *RVVTS* have so far only been demonstrated on



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

GLSVLSI '26, Canandaigua, NY, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2431-2/26/06

<https://doi.org/10.1145/3787109.3815255>

<sup>1</sup>Putting in concrete numbers for *Ara* with  $VLEN = 4096$  bits, *vtype* set to  $SEW = 8$  bit,  $LMUL = 1$  (no grouping), we get  $VLMAX = \lfloor 1 \times 4096 / 8 \rfloor = 512$ , leading to  $(512 + 1) \times (512 + 2) / 2 = 131, 841$  distinct pairs; and this is only for a single *vtype* configuration.

high-level models, including *Virtual Prototypes* (VPs), *Spike* (the reference model from *RISC-V International* [11]), and *QEMU*, because *RVVTS* currently lacks support for *Register Transfer Level* (RTL) designs; and (ii) the only other available open-source alternative, *RISC-V Vector Tests Generator* from *CHIPS Alliance* fails to reach adequate functional coverage.

**Contributions.** This paper makes the following three main contributions:

- (C1) **Evidence of insufficient functional coverage in existing open-source RVV test generator:** We present an in-depth evaluation of the *RISC-V Vector Tests Generator* from *CHIPS Alliance*, the default framework for RVV testing in *Sail-RISC-V* [10], and demonstrate that it achieves only low functional coverage and does not target negative testing.
- (C2) **Extension of RVVTS to RTL verification:** We extend the *RVVTS* framework with full RTL support, enabling its application to RVV implementations as *Design Under Tests* (DUTs).
- (C3) **Automated categorization of RVV failure modes:** Due to the high coverage of the *RVVTS* test sets and the complexity of advanced RVV RTL implementations, we expect a large number of potential failures during co-simulation. Hence, we introduce a novel rule-based *Automated Failure Categorization* (AFC) stage that automatically minimizes and clusters detected failures into failure modes, i.e., equivalence classes of mismatching tests sharing the instruction context and a minimal predicate over RVV parameters. This significantly reduces debugging effort and accelerates root-cause analysis.

In our experimental evaluation, we use the extended *RVVTS* framework on *Ara* as a strong example DUT. We generate and apply RVV test sets to *Ara* with  $VLEN = 4096$ , encompassing positive, negative, and mixed test scenarios, and achieve a functional coverage of over 96%. We then employ AFC to analyze the *Ara* failures, automatically minimizing 97% of detected failures and associating them with 611 instructions across 16 failure mode categories. Due to the large number of detected failures, a detailed per-failure analysis is beyond the scope of this paper.

To support future RVV verification research, the *RVVTS* framework extended with RTL support and AFC (including categorization rules), together with all results for *Ara*, is available in a reproducible open-source form on GitHub<sup>2</sup>.

## 2 Related Work

Automated test generation for processors is a widely researched area. Traditional methods typically decouple the architectural description from test generation, leveraging techniques such as constraint solving [13, 16]. To achieve high coverage, [19] proposed a method to construct a coverage model using constraints that describe the execution paths of individual instructions. An alternative approach employing Bayesian networks for coverage modeling was introduced in [20]. In addition, machine learning-based methods [14, 24] and fuzzing-based approaches [17, 23, 26, 34] have also been explored. However, these approaches focus exclusively on positive testing, and require significant manual effort to analyze failing tests and identify root causes. Furthermore, none of these approaches directly support RVV.

<sup>2</sup>[https://github.com/ics-jku/RVVTS\\_RTL\\_AFC\\_Ara](https://github.com/ics-jku/RVVTS_RTL_AFC_Ara)

Specifically for RISC-V, several instruction stream generators and test frameworks have been developed. One of the earliest examples is *RISC-V Torture Test Generator* [6], a random instruction generator well-suited for scalar RISC-V instructions, but not supporting RVV version 1.0. The official *Architecture Test Special Interest Group* provides architectural tests [5], including the *RISC-V Compliance Test Generator*, which generates tests for most RISC-V extensions. However, it does not cover RVV. Also approaches leveraging symbolic execution for test generation have been proposed [15, 22], but they only perform positive testing and do not support RVV. *RISCV-DV* [8], originally developed by Google and now maintained by the *CHIPS Alliance*, supports RVV but only up to version 0.9<sup>3</sup>. *FORCE-RISCV* [3], maintained by the *OpenHW Group*, supports RVV 1.0. However, it requires highly invasive modifications to the DUT, produces only raw traces, and does not include any form of result analysis as shown in [33]. Also provided by the *CHIPS Alliance* is the *RISC-V Vector Tests Generator* [7], which supports RVV 1.0 and serves as the default framework for RVV testing in *Sail-RISC-V* [10]. However, it lacks negative testing and automated result analysis, and achieves low functional coverage as demonstrated later in Section 5.1.

There are commercial solutions for RVV verification. For example, *ImperasDV* [4, 35], integrated with major *Electronic Design Automation* (EDA) simulators, provides a reference-model-based processor DV solution with the *RVVI* interface and the *riscvISACOV* SystemVerilog functional-coverage library, which is delivered as commercial Verification IP. IP vendors such as Andes Technology report using *RISCV-DV*, co-simulation against an *Instruction Set Simulator* (ISS), and commercial tools (e.g., VCS/Verdi) to enhance verification coverage for their *NX27V* RVV core [1], but the overall environment and test suites remain closed and highly project-specific. To summarize, these solutions are proprietary and leverage commercial simulators, reference models and coverage IP.

Regarding PULP *Ara*, **prior verification efforts** [25] focused on an early implementation of the vector unit with RVV version 0.7.1. These efforts specifically targeted the ***Ara* vector unit only** and employed *Universal Verification Methodology* (UVM), *RISCV-DV*, co-simulation with *Spike*, and SystemVerilog assertions. **In this work, we focus on the *Ara* vector processor**, which supports RVV 1.0 and is fully integrated with the *CVA6* core as a black-box RTL DUT. The *Ara* repository currently provides 1028 hand-crafted RVV test cases, along with a set of RVV benchmarks, offering only basic functional validation [2].

To the best of our knowledge, the extended *RVVTS* framework presented in this paper is the only open-source test framework that offers end-to-end automated RTL testing, from high functional coverage positive/negative instruction generation to result analysis and failure mode categorization, with full support for RVV 1.0.

## 3 Preliminaries

This section provides background on the the *RVVTS* framework and the PULP *Ara* RVV vector processor considered in this work.

<sup>3</sup>Compared to the draft RVV 0.9 specification, the ratified RVV 1.0 version introduces hundreds of spec changes, including a completely revised family of vector load/store instructions and a different vtype layout and vector configuration semantics. Consequently, a different verification campaign is needed.

### 3.1 The RVVTS Framework

In this work, we employ the recently introduced open-source *RVVTS* framework [30] for systematic verification of RVV. We select *RVVTS* due to its demonstrated ability to uncover previously unknown bugs in widely used simulators, including three bugs in the SystemC-based *RISC-V VP++* [31–33] and two in the *QEMU* emulator [30].

*RVVTS* supports both positive and negative testing strategies and automates the verification process from test generation to failure analysis. The framework introduces *Single Instruction Isolation* and *Code Minimization*, which reduce the manual effort required to analyze failing tests by isolating the root-cause instruction and generating minimized test cases. The automated verification flow includes grammar-based, coverage-guided test generation, instrumentation and build, functional coverage measurement, execution on a reference simulator (*Spike*) and the DUT, detection of architectural state deviations (i.e., potential failures), and subsequent failure minimization. In addition to automated execution, *RVVTS* provides interactive support through Jupyter notebooks to assist users in tracing detected failures.

### 3.2 The Ara RISC-V Vector Processor

*Ara* is a 64-bit open-source RISC-V vector processing unit, first introduced in [18] and later refined in [27, 29]. It was developed as part of the *PULP platform* and is seamlessly integrated with the open-source Ariane *CVA6* RISC-V RV64CG core [36]. The most recent *Ara* implements RVV in version 1.0 and supports multiple vector lanes, enabling parallel processing of slices of RVV registers. The implementation allows for customization of both the number of lanes and the vector register length (VLEN), with a default setting of 4 lanes and a VLEN of 4096 bits.

The *Ara* unit is modeled at the RTL level using *SystemVerilog* and can be simulated alongside the *CVA6* core using either commercial simulators (e.g., ModelSim/Questasim) or the open-source tool *Verilator* [12], which is utilized in this work. *Verilator* translates RTL code into C++, which is subsequently compiled into a single statically linked binary, referred to as the *verilated* model. This approach enables high-performance, cycle-accurate simulation and is widely adopted in academia and the open hardware community.

Prior verification efforts of *Ara* focused on an early vector unit implementation with RVV version 0.7.1, using UVM, *RISCV-DV*, co-simulation with *Spike*, and SystemVerilog assertions [25]. The current *Ara* repository provides 1028 hand-crafted RVV test cases and a set of RVV benchmarks, offering only basic functional validation [2].

## 4 Extending RVVTS

In this section, we present the extensions made to the *RVVTS* framework. First, in Section 4.1, we discuss the integration of a RVV RTL model as DUT in *RVVTS*, using *Ara* as a representative example. Then, in Section 4.2, we introduce our novel AFC stage, which analyzes *Machine State* deviations to cluster failed cases into meaningful failure mode categories.

### 4.1 RTL Model as Design Under Test

Supporting a new DUT in *RVVTS* requires implementing a corresponding *Execution Runner*, which takes a compiled program (ELF

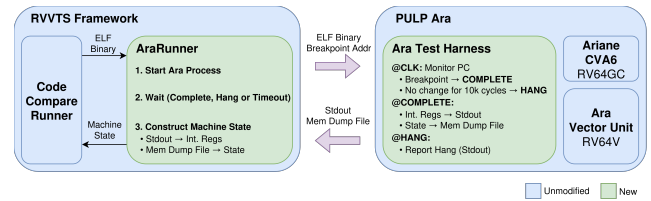


Figure 1: RVVTS RTL DUT: *AraRunner* and *Ara Test Harness*

binary) as input, executes it on the target, extracts the resulting architectural state and returns it as output. The architectural state is finally represented by the *RVVTS Machine State* data structure, which includes registers (integer, floating-point, vector, and CSR), a trap counter, the last executed *Program Counter* (PC) address, and hashes for memory areas. In this work, we use the *Ara* RTL model as a strong example DUT to demonstrate the integration of an RTL design into the *RVVTS* verification flow.

For the *Ara* RTL model we create the new *AraRunner*. To execute programs on *Ara*, the *AraRunner* utilizes the verilated *Ara* RTL model, as described in Section 3.2. In addition, to meet the requirements of *RVVTS*, a small test harness for the *Ara* RTL model is required; we build this harness based on the existing *Ara* Top-Level and introduce some modifications. Please note that, although *AraRunner* is specific to *Ara*, it can be easily adapted for other RTL models.

Figure 1 illustrates the complete integration stack of *Ara* as an *RVVTS* DUT. The blue blocks indicate the unmodified components, such as the *RVVTS* framework and all main parts of the *Ara* model except for the test harness. The green blocks denote the newly introduced components: the *AraRunner* within *RVVTS* on the left and the *Ara* test harness within the *Ara* model on the right.

The text in the green blocks describes the added behavior. We now discuss the *AraRunner* and the *Ara* test harness in more detail, by explaining the execution process. The process flow begins on the left-hand side of Figure 1 within *RVVTS*, moves to the right to the *Ara* model, and then returns back to *RVVTS*.

For each test to be executed, *RVVTS* first calls the *AraRunner* to run an ELF binary on the DUT and retrieve the resulting *Machine State*. Besides the test code, the ELF binary already includes instrumentation code generated by *RVVTS*, which extracts major parts of the *Machine State* from the target and stores it in the target’s memory before the end of execution. The target PC address, which indicates the end of execution and is referred to as the *breakpoint*, is provided through the global *RVVTS* configuration.

Next, the *AraRunner* starts the *Ara* model as new process with the ELF binary and the *breakpoint* as arguments. The *Ara* model, shown on the right-hand side of Figure 1, loads the ELF binary and begins executing the program. During execution, the *Ara* test harness monitors the PC on each clock cycle to check: (i) whether the PC has reached the *breakpoint*, and (ii) whether the PC has remained the same for 10k cycles (*hang*).

The former *breakpoint* case indicates that the program execution completed successfully. The latter *hang* case is introduced to address an observation in the current implementation of *Ara*: certain instruction sequences may cause the *CVA6* core to lock up. While

the issue is reproducible with specific instruction sequences, we are currently unable to identify a systematic pattern in these sequences.

Pseudocode of the *Ara* test harness handling the *breakpoint* and *hang* cases is shown in Figure 2. Initialization takes place in Lines 2-4. During initialization, the test harness sets up variables for breakpoint and hang detection, initializes the hang tracking structures for each commit port, clears the *Ara* memory, and loads the given ELF binary into memory.

The *breakpoint* and *hang* detection logic is executed on each positive clock edge of *Ara* (Line 9). Since *Ara* inherits the CVA6 microarchitecture, which supports retiring up to two instructions per cycle, the design provides multiple commit ports to report simultaneous retirements. Therefore, the test harness has to monitor the PC values of all commit ports (Line 11).

The *breakpoint* detection and handling logic, executed for each commit port on every positive clock edge, is shown in Line 14-26 of Figure 2. First, the harness checks whether the commit port reports a PC equal to the configured breakpoint address and whether the corresponding instruction has been acknowledged but not yet executed. Due to the instrumentation scheme used by *RVVTS* for architectural state extraction, the same breakpoint must be reached twice during execution. On the first occurrence (Lines 17-19), the harness extracts the integer register file, which is still unmodified by the *RVVTS* instrumentation code, and prints it to the standard output while allowing execution to continue. During the subsequent execution phase, the inserted *RVVTS* instrumentation writes the remaining architectural state to memory. When the breakpoint is reached a second time (Lines 20-23), the harness writes the *Ara* memory to a dump file and terminates the simulation, making the stored state available for *RVVTS*.

The *hang* detection and handling logic is shown in Line 28-39 of Figure 2. For each commit port, the harness checks whether the PC remains unchanged compared to the previous clock cycle. If the PC does not change, a corresponding hang counter is incremented. Once the hang counter exceeds the configured threshold (10k cycles), a *hang* condition is assumed. In this case, the *Ara* test harness prints a corresponding message to the standard output and terminates the simulation.

After termination of the *Ara* model, either on *breakpoint* or *hang*, the process flow is back at the *AraRunner* in *RVVTS*, left in Figure 1. At this stage, the *AraRunner* first checks the output of the run for a *hang* message. If a *hang* is detected, the *AraRunner* creates a clean new *Machine State* with the last executed PC set to an invalid value to indicate a failed execution. This *Machine State* is then returned back to *RVVTS*. Using this approach, instead of directly reporting an execution error, enables the *RVVTS* framework to handle *hang* cases in a manner similar to other *Machine State* deviations, thereby allowing the automated result analysis methods of *RVVTS* to be applied without modification, as demonstrated later in Section 5.3.

If, on the other hand, the execution completes successfully (w/o *hang*), the *AraRunner* extracts the state of the integer registers from the run's output and utilizes an existing *RVVTS* helper component to extract the remaining state from the generated memory dump file. Finally, the *AraRunner* merges these two results to construct a complete *Machine State*, which is returned to *RVVTS*, marking the end of the process flow.

```

1 // Input: elf_filename, breakpoint_addr, hang_threshold (10000)
2 int breakpoint_hits = 0;
3 int hang_pcs [CVA6Cfg.NrCommitPorts] = {-1, -1};
4 int hang_cnt [CVA6Cfg.NrCommitPorts] = {0, 0};
5
6 zero(Ara.memory); // Set all memory to zero
7 elfload(Ara.memory, elf_filename); // Load ELF binary
8
9 always @(posedge clk_i) begin
10 // Ara has multiple commit ports (default 2) -> We have to check all
11 for (int cpidx = 0; cpidx < CVA6Cfg.NrCommitPorts; cpidx++) begin
12     commit_port = Ara.commit_ports[cpidx];
13
14     // Check for breakpoint hits on commit port
15     if (commit_port.pc == breakpoint_address &&
16         commit_port.ack && !commit_port.executed) begin
17         if (breakpoint_hits == 0) begin
18             // First breakpoint hit -> Print integer registers and continue
19             $display(Ara.regfile);
20         end else if (breakpoint_hits == 1) begin
21             // Second breakpoint hit -> Create mem dump and stop (COMPLETE)
22             write_memdump(Ara.memory, "mem_dump.bin");
23             $finish();
24         end
25         breakpoint_hits = breakpoint_hits + 1;
26     end
27
28     // Check for hang on commit port
29     if (commit_port.pc == hang_pcs[cpidx]) begin
30         // PC not changed -> keep track of cycles
31         hang_cnt[cpidx] = hang_cnt[cpidx] + 1;
32         if (hang_cnt[cpidx] >= hang_threshold) begin
33             // Threshold reached -> Print "Hang" and stop (HANG)
34             $display("HANG");
35             $finish();
36         end
37     end else begin
38         hang_pcs[cpidx] = 0; // PC changed -> reset counter
39     end
40     hang_pcs[cpidx] = commit_port.pc; // Keep track of PC
41 end
42 end

```

Figure 2: *RVVTS* RTL DUT: *Ara* Test Harness (Pseudocode)

While the described integration is implemented for the *Ara* RTL model, the overall approach is not specific to *Ara*. The concept of combining a lightweight RTL test harness with an *RVVTS Execution Runner* can be applied to other RTL-based DUTs with only minor adaptations. In particular, the required functionality—program loading, execution monitoring, and architectural state extraction—can typically be implemented by extending the existing Top-Level of the respective RTL design.

## 4.2 Automated Failure Categorization (AFC)

As highlighted in the introduction, the combinatorial growth of the RVV configuration space presents significant challenges for testing. Although the *Single Instruction Isolation* method implemented in *RVVTS* already categorizes failures based on the isolated instruction, the vast state space still leads to a diverse range of **failure modes**. We define a failure mode as an equivalence class of mismatching tests that share the same instruction context, a minimal predicate over RVV parameters (e.g., SEW, LMUL, vl, vstart), and exhibit the same architectural symptom.

Concrete examples of such failure modes include: (i) **Instruction Validity Errors**: Instructions executed in unsupported configurations, e.g., reserved vtype encodings or misaligned RVV registers (leading to traps); (ii) **Vector Length and Type Handling**: Incorrect handling of vl, register grouping, or element types, including incorrect vtype updates; (iii) **Register Value Deviations**: Differences in RVV register contents after instruction execution; (iv)

**Control and Status Register Deviations:** Unexpected updates to CSRs such as vs, fs, or vtype bits; (v) **Deadlocks or Hangs:** Execution stalls that prevent test completion, detectable through invalid PC or other architectural state inconsistencies; (vi) **Cross-Instruction Propagation:** Systemic issues, e.g., in RVV register commit or unit communication, which cause multiple instructions to fail in similar ways. These failure modes – distinct from the underlying RTL root causes – must therefore be identified and analyzed systematically.

Our novel AFC stage addresses this challenge by analyzing the extracted *Machine State* from the reference simulator and the DUT, along with their deviations, to cluster failing cases into meaningful failure mode categories. Within each category, failure cases are still sub-categorized based on the isolated instruction. For each failed test identified by *RVVTS*, AFC analyzes the corresponding *Machine States* using predefined categorization rules and assigns the failure to a specific failure mode category. The categorization operates as follows: (i) The predefined category rules are evaluated sequentially. (ii) The first rule that matches determines the failure mode category for the case, ensuring that each failure is assigned to exactly one category. (iii) If no rule matches (fall-through), the case is categorized as *UNKNOWN*.

In total, we define rules for 29 failure mode categories, including the fallback category *UNKNOWN*. These categories are derived systematically from the previously introduced failure modes and capture distinct classes of architectural deviations encountered during testing. Due to space limitations, it is not feasible to describe all categories in detail within this paper. Instead, we present selected representative examples in the following.

Two representative rules for **Instruction Validity Errors** are those defining the failure mode categories *EXC\_INVALID\_ACCEPT* and *EXC\_INVALID\_REJECT*. As described earlier, the behavior of RVV instructions, including their validity, is highly dependent on the dynamic configuration. The rule for *EXC\_INVALID\_ACCEPT* checks whether the number of traps observed during execution on the reference simulator is *higher* than on the DUT, i.e., the DUT has accepted instructions that are forbidden according to the reference. Conversely, the rule for *EXC\_INVALID\_REJECT* covers cases where the number of observed traps on the reference simulator is *lower* than on the DUT, i.e., the DUT has rejected instructions that are allowed according to the reference. Both rules ignore other *Machine State* deviations, since such deviations would be a consequence of the trapping behavior: an executed instruction may have side effects (e.g., modifying register values), whereas a trapping instruction would not.

A representative rule for **Register Value Deviations** is the one defining the *IREG\_ONLY* category. This rule checks whether all *Machine State* deviations are exclusively related to the values of the 32 integer registers (including the zero register). Cases in the *IREG\_ONLY* category may, for example, indicate issues in instruction implementations (e.g., incorrect calculations) or highlight instructions that modify integer registers even though they are not specified to do so. Similar rules exist for exclusive deviations in floating-point (*FREG\_ONLY*) registers and RVV registers (*VREG\_ONLY*).

Together, the 29 defined rules allow the AFC stage to systematically classify failing tests into meaningful failure mode categories,

**Table 1: Test Set generated with RISC-V Vector Tests Generator**  
(git hash 841141b775)

Test Case Programs	Million Instructions	Million RVV Instr.	Functional Coverage (riscvOVPsim RVV)	
			Points	Percent
2,391	1,703.06	5.49 (0.32%)	3,653 / 33,076	11.04

significantly simplifying the analysis of large failure sets. In Section 5.3, we demonstrate the effectiveness of this approach through an extensive evaluation using the *Ara* vector processor as a representative RTL DUT.

## 5 Experimental Evaluation

In this section, we demonstrate the effectiveness of the extended *RVVTS* framework and our novel AFC stage by applying them to the *Ara* RTL model. First, in Section 5.1, we evaluate the test set generated by the open-source *CHIPS Alliance RISC-V Vector Tests Generator*, the default framework for RVV testing in *Sail-RISC-V*, as a baseline. Then, in Section 5.2, we introduce the two new test sets generated by *RVVTS* for *Ara*, along with the high-level results obtained for *Ara*. After this, in Section 5.3, we use AFC to provide a more detailed analysis of the results.

### 5.1 RISC-V Vector Tests Generator Test Set

In this section we evaluate the test set generated by the open-source *RISC-V Vector Tests Generator (CARVVTG)* [7] as baseline for comparison with the *RVVTS* test sets. From configurations for each RVV instruction, the *CARVVTG* framework generates a series of independent, self-checking tests as executable programs to be run on the DUT. Targeting *Ara*, we configure the framework to generate a test set for RV64GV with a *Vector Register Length (VLEN)* of 4096 bits. Furthermore, to enable the application of the same functional coverage metric used by *RVVTS (riscvOVPsim* [9]), we modify the generator to ensure that no parts of the test programs are skipped following a detected failure. This guarantees that all instructions included in the test cases are executed and accounted for by the coverage metric.

*CARVVTG* generates multiple test programs for each instruction, with each program containing tens of thousands of test cases following the same pattern but with varying values and configurations. Table 1 summarizes the major characteristics of the generated test set, including the functional coverage achieved. Overall, *CARVVTG* generates 2,391 test programs with a total size of 43 GiB. What stands out immediately is that, despite the test set including over 1.7 billion instructions, only 0.32% are RVV instructions. This is explained by the substantial overhead caused by the self-checking mechanism for each test case. However, the absolute number of 5.49 million RVV instructions is still substantial. What is a major weakness though is the rather low functional coverage of 11.04%. This is explained by the fact, that *CARVVTG* uses mostly the same code patterns and register selection sets for all test cases. Examples for this can be seen in Figure 3. The listings (left and right) show all concrete generated instructions for the vector widening add *vwadd.vv* (left) and the vector widening sum reduction *vwredsum.vs* (right) included in the test set (excluding vector mask variants). We observe that, in both cases, the concrete values for the destination

**Table 2: Test Sets generated with *CovGuidedTestsetGenerator* and applied on *Ara* (git hash a6436df6ad)**

Test set	Million Instructions	Million RVV Instr.	Functional Coverage ( <i>riscvOVPsim</i> RVV)		Detected Fails	Minimized Cases		Isolated Failing Instructions	
			Points	Percent		#Cases	Percent	Overall	Exclusive
<i>Valid Sequences</i> (VS)	2.17	0.95 (43.78 %)	31,403 / 33,076	94.94	48,973	46,601	95.16	584	13
<i>Invalid+Valid Sequences</i> (IVS)	2.04	1.00 (49.02 %)	31,950 / 33,076	94.60	33,630	33,391	99.29	598	27
<i>Merged Sequences</i> (MS) (VS + IVS)	4.21	1.95 (46.32 %)	31,951 / 33,076	96.60	82,603	79,992	96.84	611	N/A

```

// Vector Widening Int Add
    vd vs2 vs1
vwadd.vv v2, v5, v4
vwadd.vv v4, v10, v8
vwadd.vv v8, v20, v16
vwadd.vv v27, v1, v14
vd: {v2, v4, v8, v27}
vs2: {v4, v8, v14, v16}
vs1: {v1, v5, v10, v20}

// Vector Widening Int Sum Reduction
    vd vs2 vs1
vredsum.vs v1, v5, v26
vredsum.vs v4, v12, v28
vredsum.vs v6, v14, v4
vredsum.vs v24, v8, v16
vd: {v1, v4, v6, v24}
vs2: {v5, v8, v12, v14}
vs1: {v4, v16, v26, v28}

```

**Figure 3: CARVVTG Instruction Patterns**

and source registers are selected from sets containing only four elements (sets in lower part of both listings in Figure 3). Since the coverage metric awards points for all possible register combinations, large parts of the coverage points are never hit.

Another weakness of *CARVVTG* related to the restricted register selection sets, is the lack of support for testing register group overlaps, which is also mentioned in the framework’s documentation. RVV has very specific rules, when it comes to overlapping of source and destination registers. For example, overlap is explicitly allowed by RVV for the vector widening sum reduction `vwredsum.vs v0, v0, v1`. In contrast, the vector widening add `vwadd.vv v0, v0, v1` is not allowed (trap), because writing double width target elements in `v0` can interfere with the subsequent single width source elements in the same register. Both types of instruction patterns, as exemplified in Figure 3 for `vwadd.vv` and `vwredsum.vs`, are explicitly excluded by the *CARVVTG* generator. As a result, any related bugs remain undetectable.

The final major weakness of *CARVVTG* that we address is its complete lack of support for negative testing, the significance of which will be demonstrated later in Section 5.2 and Section 5.3. In negative testing, we feed a known invalid instruction to the DUT and verify whether it is correctly rejected with a trap. One example is the previously mentioned invalid `vwadd.vv v0, v0, v1` to test whether group overlap is implemented correctly and does not allow more than specified. While the test programs generated by *CARVVTG* include trap handlers, they automatically treat traps as failures. The inverse case, where a trap is expected and the absence of a trap is considered a failure, is not accounted for in the test programs.

For *CARVVTG*, we can therefore conclude that (i) functional coverage is very low at 11.04% due to the reasons outlined above, and (ii) the framework lacks support for negative testing. In the following sections, we will demonstrate that extended *RVVTS* provides significant improvements in these aspects.

## 5.2 RVVTS Test Sets and Application on *Ara*

*RVVTS* already comes with pre-generated, high-functional-coverage (>94%) test sets for positive and negative testing of RV32 and RV64 RVV implementations with a VLEN configuration of 512 bit [30]. However, *Ara* is by default configured with a VLEN of 4096 bit which prevents application of these pre-generated test sets. Although *Ara* can be reconfigured to a VLEN of 512 bits, we keep the default configuration, as it appears to be the most thoroughly tested, given that it was used in the tape-out of the *Yun System-On-Chip* (SoC) [28].

Since the existing pre-generated *RVVTS* test sets are not compatible with *Ara*, we generate new test sets: We configure *RVVTS* for RV64 and a VLEN of 4096, and generate two test sets. For the first test set, *RVVTS* is configured to generate only code sequences that don’t trigger traps when executed on the reference simulator. This results in a test set that contains only *Valid Sequences* (VS), targeting pure positive testing. For the second test set, the generator is configured to allow code sequences that trigger traps, resulting in a test set that contains *Invalid+Valid Sequences* (IVS), targeting positive/negative testing.

The left-hand side of Table 2 summarizes the major characteristics of the test sets VS and IVS, as well as the merged test set MS (i.e., VS + IVS). Both VS and IVS include more than 2 million instructions. Compared to the *CARVVTG* test set evaluated in Section 5.1, which contains 1.7 billion instructions (Table 1), the total number of instructions in VS and IVS is substantially smaller. However, the fraction of RVV instructions in the *RVVTS* test sets exceeds 43%, in contrast to only 0.32% in *CARVVTG*. As a result, each test set contains approximately one million RVV instructions, compared to 5.49 million in the *CARVVTG* test set. Although the VS and IVS test sets contain fewer RVV instructions, each achieves a functional coverage of over 94%, which is substantially higher than the 11.04% obtained with *CARVVTG*. Additionally, it is important to note that only the VS test set is directly comparable to *CARVVTG*, since *CARVVTG* does not generate invalid sequences and therefore does not support negative testing. In summary, the test sets generated by *RVVTS* achieve significantly higher coverage with far fewer instructions and, unlike *CARVVTG*, also include negative testing, which as we will see shortly, is essential.

The right-hand side of Table 2 presents the results obtained when applying the test sets to *Ara* via *RVVTS*. For VS, IVS and MS, the table shows, from left to right, the number of detected failure cases, the fractions of these cases where *RVVTS* is able to minimize them, and the number of unique instructions that are isolated by *RVVTS* as cause the failures. We see, that *RVVTS* is able to minimize the majority (>95%) of all these fails. The right-most sub-column of Table 2 presents the number of isolated unique instructions detected

**Table 3: Top 6 Failure Categories on Ara** (git hash a6436df6ad)

Failure Category (out of 29 categories)	#Detected Fails			#Minimized Cases in MS (% w.r.t fails in category)	#Instructions	Isolated Failing Instructions in MS RVV Instruction Classes		
	VS	IVS (% w.r.t. all failed cases)	MS			Class	#Cases	#Instr
<i>VREG_ONLY</i>	34,315	3,654	<b>37,969</b>	37,108 97.73%	448	integer	10,405	132
	70.07%	10.87%	<b>45.97%</b>			mask	10,290	13
<i>VTYPE_VILL_SET_ERROR</i>	90	19,858	<b>19,948</b>	19,948 100.00%	1	fixed-point	6,324	32
	0.18%	59.05%	<b>24.15%</b>			floating-point	4,135	89
<i>ARA_HANG</i>	8,384	1,606	<b>9,990</b>	8,521 85.30%	291	permutation	3,650	10
	17.12%	4.78%	<b>12.09%</b>			load	1,404	104
<i>MSTATUS_EXT_DUT</i>	6	4,303	<b>4,309</b>	4,297 99.72%	101	fixed-point	1,349	134
	0.01%	12.80%	<b>5.22%</b>			mask	1,204	6
<i>EXC_INVALID_REJECT</i>	1,174	1,242	<b>2,416</b>	2,415 99.96%	65	mask	514	2
	2.40%	3.69%	<b>2.92%</b>			integer	298	9
<i>EXC_INVALID_ACCEPT</i>	3	2,309	<b>2,312</b>	2,310 99.91%	413	floating-point	57	22
	0.01%	6.87%	<b>2.80%</b>			reduction	45	4
<i>OTHERS</i>	5,001	658	<b>5,659</b>	5,393 95.30%	155	floating-point	3,277	91
	10 categories w/ cases	10.21%	1.96%			<b>6.85%</b>	reduction	589
13 categories w/o cases						permutation	431	4
0 cases in <i>UNKNOWN</i>						reduction	1,043	16
						mask	825	10
						integer	511	27
						floating-point	13	8
						permutation	11	2
						store	7	1
						load	5	1
						floating-point	750	91
						load	367	107
						integer	332	92
						fixed-point	240	32
						permutation	199	15
						store	164	63
						reduction	148	8
						mask	110	5
						store	2,076	124
						Zicr config	1,714	2
						permutation	1,104	2
						mask	382	2
						config	55	2
						floating-point	46	6
						fixed-point	9	7
						load	5	5
						reduction	2	2

exclusively by the given test set, which is calculated e.g. for IVS by subtracting the number of isolated instructions from VS from MS, i.e.,  $611 - 584 = 27$ . These values highlight the importance of positive and negative testing: Since instruction sequences containing illegal parameter combinations or configurations are, by definition, not included in the VS test set, bugs related to such invalid sequences can only be detected through negative testing and, therefore, with the IVS test set. Conversely, since VS, which targets pure positive testing, is roughly the same size as IVS but does not contain invalid sequences, it includes a larger number of valid sequences and can, therefore, detect cases not covered by IVS. In the next section, we will utilize AFC to conduct a more detailed analysis of the results.

### 5.3 Automated Failure Categorization on Ara

As shown in the previous section, *RVVTS* is able to trace the majority of failures down to individual instructions. However, analyzing the root causes of a large number of failures per instruction still requires significant manual effort. To tackle this challenge, we next apply our AFC stage, presented in Section 4.2, to provide a more detailed analysis of the results.

The results generated by the applied AFC, are shown in Table 3. The table rows list the top 6 failure mode categories, ranked by the number of failures and accounting for more than 93% of all failed cases. The final row, *Others*, summarizes the failure mode categories not included in these top 6. The second column presents the number of detected fails for the VS, IVS and the merged MS test set. From the numbers for VS and IVS we can observe whether failures are found

rather by a pure positive (VS) or by a positive/negative (IVS) testing strategy. The following columns present the numbers of cases that are minimized by *RVVTS* and the number of isolated unique instructions. The final column presents the isolated instructions, grouped into instruction categories based on the RVV spec., along with number of cases and number isolated instructions, ranked by number of cases. We will now briefly discuss the 6 top failure mode categories presented in Table 3.

The top-most category presented in Table 3 is *VREG\_ONLY*. This category contains all cases with deviations exclusively in the values within RVV registers (i.e. there must be no deviations in other registers, CSRs, etc.). Cases in this category may, for example, indicate issues in instruction implementations (e.g., incorrect calculations) or in RVV register handling. As shown in Table 3, such cases are predominantly identified through positive testing on *Ara* (~70% for VS compared to ~11% for IVS).

The second category, *VTYPE\_VILL\_SET\_ERROR*, contains cases related to the *vill* bit in the *vtype* CSR, which indicates that the RVV configuration is illegal, causing subsequent vector instructions to trap. More specifically, this category includes cases where the *vill* bit is set on the reference simulator, but not on DUT. For *Ara*, all 19,948 detected cases in this category are related to the exact same issue, namely that the *vill* bit is not writable using the `vsetvl` configuration instruction. As shown in Table 3, cases in this category are primarily identified via negative testing (IVS).

The third category, *ARA\_HANG*, consists of all cases where a hang is detected in *Ara*, preventing the execution of a test case from

completing, as described in Section 4.1. The cases are identified by checking for an invalid value for the last executed PC in the DUT *Machine State*. Interestingly, cases in this category are primarily identified via positive testing (VS) and the majority of cases can be minimized. This indicates issues in instruction processing and execution, rather than in instruction validity checks.

The fourth category, *MSTATUS\_EXT\_REF*, contains cases related to the vs and fs bits in the mstatus CSR, which represent the enable and dirty bits for the RVV and floating-point extensions. More specifically, this category includes cases where the number of set bits after execution on the DUT is higher than on the reference. Cases in this category may hint to instructions that invalidly enable extensions. As shown in Table 3, cases in this category are primarily identified via negative testing (IVS). With *MSTATUS\_EXT\_REF*, there is also a direct counterpart to this category, for which, however, no fail cases were found on *Ara*.

The two last categories, *EXC\_INVALID\_REJECT/ACCEPT*, are also counterparts to each other. In both cases, the numbers of traps triggered during execution are compared. *EXC\_INVALID\_REJECT* contains the cases where the number of traps was higher on the DUT, which indicates that the DUT rejected instructions considered valid by the reference. In contrast, *EXC\_INVALID\_ACCEPT* contains the cases where the number of traps was higher on the reference, indicating that the DUT accepted instructions considered invalid by the reference. As shown in Table 3, the latter category, in particular, is primarily identified via negative testing (IVS).

Reaching the final row, *OTHERS*, we observe that AFC successfully categorized all detected *Ara* failures into 16 (6 top categories + 10 with cases) out of a total of 29 categories. Notably, all cases are successfully categorized, as none fall into the *UNKNOWN* category.

## 6 Conclusions

We introduced an extended, open-source *RVVTS*-based framework for RTL verification of RVV 1.0, featuring a novel *Automated Failure Categorization* (AFC) stage for scalable failure analysis. Using the *Ara* vector processor as DUT, we achieved over 96% functional coverage, significantly outperforming the open-source *CHIPS Alliance RISC-V Vector Tests Generator*, which serves as the default framework for RVV testing in *Sail-RISC-V*, used here as a baseline. Our framework detected over 82k deviations, minimized 97% of them, and clustered failures into 16 categories, streamlining debugging and root-cause analysis. The extended *RVVTS* framework and all *Ara* experimental results are publicly available on GitHub.

## Acknowledgments

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

## References

- [1] 2026. AndesCore NX27V Processor – 64-bit CPU with RISC-V Vector Extension. <https://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v>.
- [2] 2026. Ara Vector Processor. <https://github.com/pulp-platform/ara>.
- [3] 2026. FORCE-RISCV RISC-V Instruction Sequence Generator (ISG). <https://github.com/openhwgroup/force-riscv>.
- [4] 2026. ImperasDV – RISC-V Processor Verification Made Easy. <https://www.synopsys.com/verification/imperasdv.html>.
- [5] 2026. RISC-V Architecture Test SIG. <https://github.com/riscv-non-isa/riscv-arch-test>.
- [6] 2026. RISC-V Torture Test Generator. <https://github.com/ucb-bar/riscv-torture>.
- [7] 2026. RISC-V Vector Tests Generator. <https://github.com/chipsalliance/riscv-vector-tests>.
- [8] 2026. RISC-V-DV. <https://github.com/google/riscv-dv>.
- [9] 2026. riscvOVPSim Imperas RISC-V Instruction Set Simulator (ISS). <https://www.imperas.com/riscvovpsim-free-imperas-risc-v-instruction-set-simulator>.
- [10] 2026. Sail RISC-V: Formal Specification of the RISC-V ISA. <https://github.com/riscv/sail-riscv>.
- [11] 2026. Spike RISC-V ISA Simulator. <https://github.com/riscv/riscv-isa-sim>.
- [12] 2026. Verilator - A fast Verilog/SystemVerilog simulator. <https://www.verilator.org>.
- [13] 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. *DTC* (2004), 84–93.
- [14] Niklas Bruns, Daniel Große, and Rolf Drechsler. 2020. Early Verification of ISA Extension Specifications Using Deep Reinforcement Learning. In *GLSVLSI* 297–302.
- [15] Niklas Bruns, Vladimir Herdt, and Rolf Drechsler. 2023. Processor Verification using Symbolic Execution: A RISC-V Case-Study. In *DATE*. 1–6.
- [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] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2023. ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance. In *International Symposium on Hardware Oriented Security and Trust*. 1–12.
- [18] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2020. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. *TVLSI* 28, 2 (2020), 530–543.
- [19] Mikhail Chupilko, Alexander Kamkin, Artem Kotsyniak, and Andrei Tatarnikov. 2017. MicroTESK: Specification-Based Tool for Constructing Test Program Generators. In *Haifa Verification Conference*.
- [20] Shai Fine and Avi Ziv. 2003. Coverage directed test generation for functional verification using Bayesian networks. In *DAC*. 286–291.
- [21] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side. In *DAC*. 1–6.
- [22] Vladimir Herdt, Sören Tempel, Daniel Große, and Rolf Drechsler. 2021. Mutation-based Compliance Testing for RISC-V. In *ASP-DAC*. 55–60.
- [23] 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.
- [24] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. 2011. Feedback-Based Coverage Directed Test Generation: An Industrial Evaluation. 112–128.
- [25] 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* 40, 3 (2023), 36–44.
- [26] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. 261–272.
- [27] Matteo Perotti, Matheus Cavalcante, Renzo Andri, Lukas Cavigelli, and Luca Benini. 2024. Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor. *TC* 73, 7 (2024), 1822–1836.
- [28] Matteo Perotti, Matheus Cavalcante, Alessandro Ottaviano, Jiantao Liu, and Luca Benini. 2023. Yun: An Open-Source, 64-Bit RISC-V-Based Vector Processor With Multi-Precision Integer and Floating-Point Support in 65-nm CMOS. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3732–3736.
- [29] Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. 2022. A “New Ara” for Vector Computing: An Open Source Highly Efficient RISC-V 1.0 Vector Processor Design. In *International Conference on Application-specific Systems, Architectures and Processors*. 43–51.
- [30] Manfred Schlägl and Daniel Große. 2024. Single Instruction Isolation for RISC-V Vector Test Failures. In *ICCAD*. 156:1–156:9.
- [31] Manfred Schlägl and Daniel Große. 2025. Fast Interpreter-Based Instruction Set Simulation for Virtual Prototypes. In *DATE*. 1–7.
- [32] Manfred Schlägl, Christoph Hazott, and Daniel Große. 2024. RISC-V VP++: Next Generation Open-Source Virtual Prototype. In *OSDA*.
- [33] Manfred Schlägl, Moritz Stockinger, and Daniel Große. 2024. A RISC-V “V” VP: Unlocking Vector Processing for Evaluation at the System Level. In *DATE*. 1–6.
- [34] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU fuzzing via intricate program generation. In *USENIX Conference on Security Symposium*. USA.
- [35] Chloe Tain, Savita Patil, and Hussain Al-Aasaad. 2025. Survey of Verification of RISC-V Processors. *JETTA* 41, 2 (May 2025), 111–138.
- [36] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *TVLSI* 27, 11 (Nov 2019), 2629–2640.