FP-RVVTS: Sail-guided Verification of RISC-V Floating-Point Implementations

Katharina Ruep, Manfred Schlägl, Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
{katharina.ruep, manfred.schlaegl, daniel.grosse}@jku.at

***Abstract*—This paper presents a Sail-guided *FP-RVVTS* flow that uses *Sail RISC-V* as an executable RISC-V specification for differential verification of RISC-V *Floating-Point* (FP) implementations, from *Instruction Set Simulator*s (ISSs) to *Register Transfer Level* (RTL) designs. *FP-RVVTS* generates positive and negative tests for RV32/RV64 F, D, and Zfh, executes them against *Sail RISC-V* and a *Design Under Test* (DUT), and compares the resulting architectural state. Observed mismatches are minimized to compact, self-contained failing test cases and characterized by affected state, flags, exceptions, special values, and precision interactions to support root-cause analysis. We evaluate ISSs, FP-library integrations, and the PULP ARA RTL implementation. The generated tests achieve very high F/D functional coverage and expose 9 non-trivial bugs, including 5 newly discovered issues. These results demonstrate that the Sail-guided *FP-RVVTS* flow enables practical RISC-V FP verification across ISSs, FP-library integrations, and RTL implementations.**

I. Introduction

Although *Floating-Point* (FP) instructions constitute only a modest fraction of an *Instruction Set Architecture* (ISA) by instruction count, their semantics make them disproportionately difficult to implement correctly. Their behavior is governed by a dense combination of finite precision, multiple rounding modes, exception flags, subnormal numbers, infinities, signed zeros, and not-a-number values (NaNs) [1]. These semantic features are central to IEEE-754-compatible numerical and data-intensive software, including scientific computing, signal processing, graphics, and machine-learning workloads. At the same time, they create corner cases in which a single result bit, exception flag, or NaN-boxed value can determine whether an implementation is architecturally correct. For RISC-V [2], [3], this challenge is amplified by an open and rapidly evolving ecosystem: simulators, software FP libraries, processor generators, and *Register Transfer Level* (RTL) implementations are developed by different communities and are often used interchangeably as reference models, validation targets, or execution platforms.

Testing FP support in this setting is not only a matter of producing many random instructions [4]. The generated programs must be architecturally valid, exercise exceptional behavior deliberately, cover interactions between the RISC-V F, D, and Zfh extensions, and preserve enough state to make failures reproducible. At the same time, a test oracle must be trusted across heterogeneous *Design Under Test*s (DUTs). Using one simulator as the golden reference can be practical, but it also risks turning simulator-specific behavior into the de facto specification. This is particularly problematic for FP instructions, where the corner cases described above often appear only in rare operand combinations and can therefore be missed by conventional testing.

979-8-3195-3997-7/26/$31.00 ©2026 IEEE

This paper therefore takes *Sail RISC-V* [5] as the golden reference for FP verification. Unlike a conventional reference simulator, *Sail RISC-V* is the Sail formal specification of the RISC-V architecture adopted by RISC-V International [5]. It defines instruction formats, encoders and decoders, and instruction semantics in a language designed for precise ISA descriptions, and can be compiled into an executable model while also serving as a basis for formal artifacts [6]. This changes the role of the oracle in the verification flow: generated tests are checked against an executable form of the architectural model rather than against another implementation that may encode its own design choices. For FP behavior, where architecturally relevant mismatches may arise only for rare operand combinations and manifest as bit-level deviations, this distinction is essential, as it enables deviations in simulators, FP libraries, and RTL implementations to be evaluated with respect to the architectural model.

The starting point of our work is *RVVTS*, an open-source framework built around a grammar-based RISC-V *Instruction Sequence Generator* (ISG) and supporting positive and negative testing, differential execution, failure isolation, and minimization [7], [8]. Recent late-breaking work introduced *FP-RVVTS* as a FP extension of this framework [9]. It added a context-free grammar for the RV32/RV64 F, D, and Zfh FP extensions and enriched the grammar with dependency annotations to derive compact test cases that reproduce observed failures. That work demonstrated high-coverage test generation for RISC-V simulators and their FP backends, including *SoftFloat* (SF) [10] and *FloppyFloat* (FF) [11]. However, it used *Spike* as the reference simulator and was limited to simulator DUTs, leaving formal specification-guided and RTL-level verification open.

Contribution: This paper advances *FP-RVVTS* in two complementary directions: toward specification-driven verification by replacing the simulator-based oracle with *Sail RISC-V*, and toward implementation-level verification by including RTL DUTs as verification targets. We integrate *Sail RISC-V* into *FP-RVVTS* as an executable architectural reference for test-set generation and differential test-set execution. The grammar-based FP ISG and dependency annotations from

prior *FP-RVVTS* work are refined and used for compact failure reduction through *Dependency-Based Test-Case Minimization*. In addition, we introduce *Automated Failure Characterization* to group failures by instruction, affected architectural state, exception behavior, FP flags, special-value handling, and precision interactions, thereby supporting root-cause analysis across simulators, FP-library integrations, and RTL targets.

We evaluate *FP-RVVTS* on a heterogeneous set of RISC-V FP implementations spanning *Instruction Set Simulator*s (ISSs), FP-library integrations, and RTL. As simulators we include *Spike* [12], *RISC-V VP++* [13], [14], *QEMU* [15], and configurations based on SF and FF, covering both simulator-level behavior and the effect of different FP backends. For the simulator configurations, the Sail-guided flow confirms the failures previously observed with a Spike-based reference, showing that these issues are not artifacts of a simulator-specific oracle but also hold with respect to the executable architectural specification. To demonstrate *FP-RVVTS* beyond simulators, we apply it to the RTL implementation of PULP ARA [16], a silicon-proven and representative high-performance RISC-V vector processor. The ARA case study exposes HW-specific FP failures, including missing *Not a Number* (NaN)-boxing checks, incorrect exception-flag behavior, and critical deviations in division and square-root operations. Overall, this demonstrates the value of and need for a Sail-guided methodology targeting the full spectrum from simulators to RTL.

The complete *FP-RVVTS* framework, including all presented extensions and evaluation artifacts for the investigated DUTs, are available as open source on GitHub[1].

II. Related Work

Verification of RISC-V processors has been addressed by a broad range of simulation-based, formal, and hybrid techniques [17]. Existing work includes compliance-oriented test generation [18], mutation-based compliance testing [19], cross-level simulation between ISSs and RTL models [20], [21], equivalent-program and metamorphic testing [22], [23], open validation suites [24], constrained-random generation such as Google's RISCV-DV [25], and recent processor fuzzing approaches [4], [26], [27]. These methods target broad ISA correctness, compliance, or RTL bring-up across many instruction classes. They may expose FP failures when such instructions occur in generated programs, but they do not make the RISC-V FP extensions, exception flags, rounding modes, NaN-boxing, or FP-library behavior the central coverage objective. For example, RISCV-DV initializes FP registers with special values at the beginning of a test, which limits operand diversity during later execution, whereas processor fuzzers such as Cascade [27] treat FP behavior as one part of a larger processor state space.

Beyond instruction generation, prior work has also studied cross-level checking and verification infrastructure for RTL targets. Cross-level testing executes generated instruction streams on an RTL core and an ISS reference [20], [21], while recent verification environments for RISC-V vector accelerators combine random instruction generation, scoreboards, and functional coverage for complex vector RTL targets [28]. Such flows are effective for exposing implementation mismatches, but their oracles are typically conventional simulators, manually constructed checkers, or design-specific scoreboards. At the specification level, Sail provides executable and mechanized ISA semantics for architectures including RISC-V [6], and *Sail RISC-V* has been adopted as the RISC-V architectural model [5]. This work uses *Sail RISC-V* as the golden reference for both test-set generation and differential test-set execution. Thus, mismatches are evaluated against an executable architectural specification rather than against another simulator or a design-specific reference model. Moreover, the Sail-guided *FP-RVVTS* flow applies one executable architectural oracle across simulators, FP-library integrations, and RTL implementations. It retains failure isolation and state minimization for compact instruction-level debugging, and adds automated failure characterization to support root-cause analysis.

FP verification has also been studied for dedicated hardware blocks and for RISC-V FP instructions. For instance, Kaiser et al. [29] verify a RISC-V-compatible fused multiply-add unit using a UVM-based environment and an external processor *Floating-Point Unit* (FPU) as reference model. A closely related RISC-V-specific work is the coverage- and constraint-based RV32F test-generation approach by Lu et al. [30], which explicitly targets FP instruction verification for one concrete RTL processor. Unlike this RV32F-specific approach, *FP-RVVTS* covers RV32/RV64 F, D, and Zfh, is available as open source, uses *Sail RISC-V* as the executable architectural model rather than a simulator or processor FPU, and targets ISSs, FP-library integrations, and RTL.

III. FP-RVVTS

FP-RVVTS automates the generation, execution, and failure isolation of FP test cases through a systematic multi-stage verification flow that maximizes coverage while minimizing manual effort. In this section, we present the complete *FP-RVVTS* verification flow and its associated extensions and enhancements.

We begin in Section III-A by introducing the two primary *FP-RVVTS* sub-flows, *Test-Set Generation* and *Test-Set Execution*. Next, Section III-B describes the integration of our new golden reference model, *Sail RISC-V*, into *FP-RVVTS*. In Section III-C, we introduce the grammar-based ISG used to randomly generate valid FP code sequences. Subsequently, Section III-D describes the *Dependency-Based Test-Case Minimization* used for compact failure reduction, and Section III-E presents our *Automated Failure Characterization* approach for accelerating root-cause analysis.

A. Primary FP-RVVTS Sub-Flows

The first sub-flow, *Test-Set Generation*, shown in Fig. 1, generates high-coverage test sets for both positive and negative testing of RISC-V FP implementations. Starting from an

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

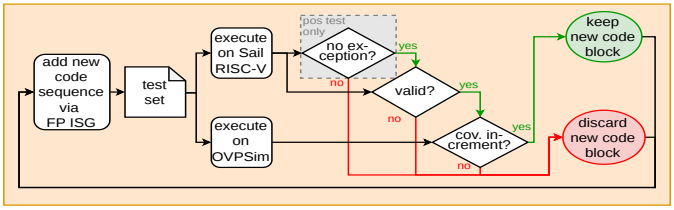

Fig. 1: Overview of FP-RVVTS's *Test-Set Generation* sub-flow.

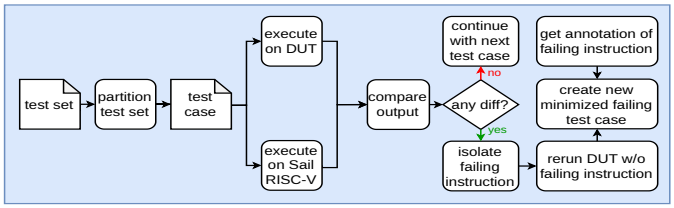

Fig. 2: Overview of FP-RVVTS's *Test-Set Execution* sub-flow.

empty test set, the grammar-based FP ISG (see Section III-C) randomly generates code sequences that are added to the test set. The resulting test set is then executed on the golden reference *Sail RISC-V* (see Section III-B) for validation and on *riscvOVPSim* [31] for functional coverage measurement, as shown by the two execution paths in Fig. 1. Based on this, *FP-RVVTS* verifies that the augmented test set remains valid and that overall coverage improves. In *pos* mode, only code sequences that do not raise exceptions are admitted, ensuring that the resulting test set targets positive testing exclusively. If all conditions are met, the new code sequences are retained; otherwise, the added code sequences are discarded and the previous state of the test set is restored. This process repeats until the test set reaches a predefined coverage threshold.

The second sub-flow, *Test-Set Execution*, shown in Fig. 2, applies a pre-generated test set to the DUT, compares outcomes to the reference model, and drives failure isolation and failing test-case minimization: The sub-flow partitions the given **test set** into small, self-contained **test cases** of configurable size. Each of these test cases is executed on the DUT and the golden reference *Sail RISC-V*. Both executions produce comparable architectural-state outputs (*Machine States*), comprising, for example, register values, a trap counter, and hashes of relevant memory regions. If differences are detected between the *Machine States* of the DUT and the reference, *FP-RVVTS* classifies the test case as failed and initiates failure isolation: First, the failing instruction is isolated as described in [7]. *FP-RVVTS* then applies our *Dependency-Based Test-Case Minimization* method, described in detail in Section III-D. The test case is re-executed without the failing instruction to obtain the *Machine State* immediately before the failure. Using this state and the dependency annotations of the failing instruction, *FP-RVVTS* synthesizes a minimized failing test case containing only the required state initialization and the isolated instruction.

B. Sail RISC-V Integration

FP-RVVTS is a modular framework structured around so-called *Runners*, which encapsulate different stages of the verification flow. Supporting a new executable reference model

```
...
"<instr_f_calc>": [
    "<instr_f_calc1><flen> <frd>, <frs1><rm>",
    "<instr_f_calc2><flen> <frd>, <frs1>, <frs2><rm>",
    "<instr_f_calc3><flen> <frd>, <frs1>, <frs2>, <frs3><rm>",
],
"<instr_f_calc2>": ["fadd", "fsub", "fmul", "fdiv"],
...
```

Listing 1: Arithmetic instruction subset of the FP grammar.

in *FP-RVVTS* requires implementing an *ExecutionRunner* that executes compiled test cases on the target model and extracts the resulting *Machine State* for use by *FP-RVVTS*, as described in Section III-A.

To integrate *Sail RISC-V* as a golden reference model, we implement the *SailRunner* as a dedicated *ExecutionRunner*. The extraction of the *Machine State* from *Sail RISC-V* is realized through two complementary mechanisms. First, *FP-RVVTS* instruments each test case with code that, immediately after the test-case section completes, stores the relevant architectural state in a dedicated memory region on the target. Second, we extend *Sail RISC-V* with a mechanism that generates a memory dump file containing the stored state before *Sail RISC-V* terminates. This memory dump file is subsequently parsed by *FP-RVVTS* to reconstruct the corresponding *Machine State* for automated state comparison, test-case minimization, and analysis.

C. Random FP Instruction Sequence Generation

In this section we introduce the grammar-based ISG, which underpins the high-coverage FP *Test-Set Generation* sub-flow presented in Section III-A.

The FP grammar is structured as a production system where non-terminals map to either string templates, instruction lists or callable generators. Configuration parameters (e.g., enabled extensions, base ISA width) decide which productions are valid, ensuring all generated instruction sequences respect the target ISA without manual retargeting. From the start symbol, the grammar-driven ISG recursively expands non-terminals, randomly selecting instructions and operands from the FP and integer register files to produce valid, architecturally compliant instruction sequences.

Example 1. *Listing 1 shows the FP grammar for arithmetic instructions. The non-terminal* `<instr_f_calc>` *expands to a template with precision* `<flen>`*, operand registers, and rounding mode* `<rm>`*. For instance,* `fadd` *is a two-operand instruction derived via* `<instr_f_calc2>`*. Resolving all non-terminals yields a concrete instruction such as* `fadd.s f1, f2, f3, rne`*.*

The full FP grammar includes production rules for load/store, arithmetic, conversion, and move instructions. Architectural parameters are set per compilation target (*RV32* or *RV64*). The chosen base ISA determines the available integer register widths used by integer-FP conversions, while the enabled FP extensions govern the available FP operations and formats. For example, with only the RISC-V *F* extension enabled, the ISG generates single-precision FP instructions

```
...
"<frd>": ("<f_reg>", {"clob": "_"}),
"<frs1>":("<f_reg>", {"dep": "_"}),
"<frs2>":("<f_reg>", {"dep": "_"}),
...
```

Listing 2: Register part of the FP grammar with dependency annotation.

and omits double-precision instructions such as `fld` and `fsd`. Any combination of *F*, *D* and *Zfh* that satisfy the architectural extension dependencies is supported.

In addition, both RISC-V rounding-mode mechanisms are supported: static selection via the `rm` field of an instruction, and dynamic selection via the `frm` field of `fcsr` when `rm` is set to `DYN`. The ISG randomizes `frm` alongside other *Control and Status Register* (CSR) state, varying rounding mode and status flag combinations across test code and improving coverage over [9].

D. Dependency-Based Test-Case Minimization

As introduced in the *Test-Set Execution* sub-flow in Section III-A, *Dependency-Based Test-Case Minimization* reduces an isolated failure to a compact, self-contained test case.

After *FP-RVVTS* has isolated the failing instruction, the instruction's dependency metadata, derived from the dependency annotations during generation, is used to determine the *Machine State* elements required to reproduce its behavior. This metadata is generated by the ISG from the FP grammar during *Test-Set Generation* and captures both dependencies (e.g., source registers, CSRs) and potential effects (e.g., destination registers). The original test case is then re-executed on *Sail RISC-V* with the failing instruction removed, yielding the pre-failure *Machine State*, i.e., the *Machine State* immediately before the failing instruction would have executed. From this state, *FP-RVVTS* extracts the values referenced by the failing instruction's metadata and synthesizes minimal state-initialization code, which is combined with the isolated failing instruction to form the minimized failing test case.

In the following, we describe how the FP grammar is annotated with dependency information and how the ISG propagates this information to instruction-level dependency metadata. Relevant grammar entries are labeled as either `dep` or `clob`. The `dep` label denotes state required to reproduce the instruction's behavior, while the `clob` label identifies state modified by the instruction whose pre-instruction value must be captured to make the failure observable.

Example 2. *Listing 2 shows a snippet of the FP grammar. The destination register* `<frd>` *is annotated* `clob`*, while source registers are annotated* `dep` *(e.g.,* `<frs1>`*). The wildcard "_" applies the annotation to any register resolved by that production rule, eliminating the need for per-register annotations. Revisiting Example 1 with the instruction* `fadd.s f1, f2, f3, rne`*, e.g., the wildcard for* `<frd>` *would be resolved to* `f1` *and* `<frs1>` *would be resolved to* `f2`*.*

Once the grammar is annotated, every *FP-RVVTS*-generated test case includes concrete `dep` and `clob` annotations for each instruction. Based on this, for each failure, *Dependency-Based Test-Case Minimization* automatically constructs a minimized failing test case containing only the required architectural-state initialization and the single failing instruction.

```
// restore floating point register
_reg_f24:.byte 0x00,0x40,0xc7,0x44,0xff,0xff,0xff,0xff
la t0, _reg_f24
fld f24, 0(t0)
// restore fcsr {'nx': F, 'uf': F, 'of': F, 'dz': F, 'nv': T, 'rm': 'rne', 'res': 0}
li t0, 0x10
csrrw zero, fcsr, t0
// restore mstatus.fs/vs = {'fs': 'dirty', 'vs': 'off'}
li t0, 0x6600
csrc mstatus, t0
li t0, 0x6000
csrs mstatus, t0
// restore registers
li x27, 0x802160ba
// INSTR. DEP. METADATA: {'dep': {'fcsr.rm', 'mstatus.fs/vs.fs', 'x27'}, 'clob': {'f24'}}
fcvt.h.l f24, x27, rne
```

Listing 3: Example of minimal snapshot for failing instruction.

Example 3. *Listing 3 shows a minimized test case. The last line contains the failing instruction* `fcvt.h.l`*, which converts the integer in* `x27` *to a half-precision FP in* `f24`*. The highlighted comment above it lists its dependency metadata:* `dep` *(CSRs, source register) and* `clob` *(destination register). To reconstruct the* Machine State *before execution,* `f24` *is restored to its pre-instruction value, followed by the CSRs* `fcsr` *and* `mstatus`*, and finally* `x27`*. The result is a compact, self-contained test case that reliably reproduces the failure, greatly aiding debugging.*

E. Automated Failure Characterization

In this section we introduce our *Automated Failure Characterization* method, which post-processes failures to summarize key attributes and accelerate root-cause analysis. For each detected failure, *FP-RVVTS* generates a report containing the instruction, rounding mode, state differences, affected registers, and attributes such as exceptions, FP flags, special values (NaN, *Infinity* (Inf), zero), and precision mismatches. Affected registers provide information about their values and reveal the precision stored in each register (e.g., `_h` for half-precision). Assigned attributes help identify similarities between failures, enabling them to be automatically grouped to find the common root causes. Together, these reports facilitate root-cause analysis by connecting observable differences to their underlying causes.

Example 4. *Listing 4 shows the report of the failed minimized test case introduced in Listing 3. This test case is part of the set that exposed bug6 in* PULP ARA*, discussed later in Section IV-B. The attributes indicate that the* Invalid Operation *(NV),* Overflow *(OF), and* Inexact *(NX) flags are set and that the special value Inf appears. In the diff section, a mismatch in the OF flag is visible (highlighted red): the REF sets the flag* `True`*, whereas the DUT does not. Notably, the destination FP register* `f24` *in the register section contains the same Inf value in both (highlighted green), indicating that the bug resides in the flag generation logic rather than the conversion itself.*

The next section evaluates *FP-RVVTS* on six diverse DUTs and analyzes the bugs uncovered by the generated test sets.

```
+=====================================================+
Instruction: fcvt.h.l f24, x27, rne
+=====================================================+
Attributes:  fcsr ['invalid', 'overflow', 'inexact']
             special values ['inf']
+-----------------------------------------------------+
diffs:
REG     REF                    DUT                DIFF
fcsr.of True                   False                 X
+-----------------------------------------------------+
Registers: f24, x27
REG  REF                       DUT
+-----------------------------------------------------+
x27  0x00000000802160ba        0x00000000802160ba
f24  0xffffffffffff7c00/inf_h  0xffffffffffff7c00/inf_h
+=====================================================+
```

Listing 4: Example report for a failure. (excerpt)

IV. Evaluation

For evaluation, we consider 12 *test setups*, formed by pairing 6 DUT configurations with 2 test sets. As DUTs we include three RISC-V simulators: *Spike* [12], *RISC-V VP++* [13], and *QEMU* [15], with FP libraries SF [10] and FF [11] integrated into the first two. Additionally, we include the RTL implementation of PULP ARA [16]. We used the *RV64IFD_Zfh* configuration to provide the broadest FP coverage supported across all tested DUTs, covering single-precision (*F*, FP32), double-precision (*D*, FP64), and half-precision (*Zfh*, FP16).

The test sets consist of a positive (*pos*) and a negative (*neg*) test set. Both are generated using the *Test-Set Generation* sub-flow presented in Section III, with *Sail RISC-V* serving as the validation model and *riscvOVPSim* used for functional coverage measurement. The positive test set comprises code that executes without exceptions, validating correct execution paths; the negative test set includes code that triggers exceptions, ensuring proper exception handling and fault detection. Coverage measurement is restricted to *F* and *D* extensions, as the *Zfh* extension is not supported by *riscvOVPSim*. Instruction coverage reaches 100%. Functional coverage reaches 99.94% with $5\,108$ test cases in the negative test set with only one case uncovered: For load/store instructions, the zero register `x0` is never used as `rs1`. Addresses for these instructions are computed beforehand in the ISG, so `rs1` always holds a valid and non-zero base address. For the positive test set, coverage reaches 98.87% with $5\,271$ test cases because reserved rounding modes (*rm*) would raise exceptions and are therefore excluded by construction.

A. Application on DUTs and Results

Using the *Test-Set Execution* sub-flow and its *Dependency-Based Test-Case Minimization*, presented in Section III, we evaluate all considered DUTs with the generated test sets. Table I summarizes the observed failures, indexed by test setup ID, DUT, and test set (*pos* or *neg*). The remaining columns list the isolated failing instructions, with each cell reporting the number of failures (architectural state differences between *Sail RISC-V* and the DUT). Both *Spike SF* (1, 2) and *QEMU* (9, 10) match *Sail RISC-V* across both test sets, thus produce no failures. This supports our choice of *Sail RISC-V* as the golden reference, since it confirms the *Spike*-based results from [9] while providing a stronger, specification-aligned foundation. The failures in the remaining 8 test setups (3–8, 11, 12) are analyzed next.

B. Failure Characterization and Bug Analysis

Using the *Automated Failure Characterization* introduced in Section III-E and similarity checks of the resulting reports, we identified the dedicated bugs summarized in Table II.

Before discussing the newly discovered bugs, we briefly revisit **bug1–4**, which were first reported in [9] using *Spike* as the golden reference. Our *Sail*-guided flow confirms that all four are genuine ISA non-conformances rather than artifacts of a simulator-specific oracle: **bug1** occurs in RISC-V VP++ SF, when at least one input is NaN, the DUT returns NaN instead of the non-NaN operand, implementing the pre-v2.2 RISC-V specification behavior rather than the current semantics introduced in v2.2. **bug2a/b** affect FP load/store instructions in RISC-V VP++, when the FP extension is disabled, RISC-V VP++ omits the required extension-enabled check and executes the instruction instead of raising an exception, whereas *Sail RISC-V* correctly signals a fault. **bug3** reveals that RISC-V VP++ FF correctly computes the NaN result but then zero-extends the 32-bit value to 64 bits instead of sign-extending it as required by the RISC-V specification. **bug4** shows an incorrectly generated OF flag, with *Sail RISC-V* setting the flag as required while the *Spike FF* does not. We now discuss the newly discovered **bug5–9** in more detail.

bug5 affects `fmv.x.h` in RISC-V VP++. When the source FP register holds an invalidly NaN-boxed FP16 value, the DUT incorrectly substitutes a canonical quiet NaN before extracting the bits, producing `0x7E00` instead of the correct sign-extended lower 16 bits. Per the RISC-V specification, `fmv` instructions are explicitly exempt from NaN-boxing checks and must transfer raw bits directly without reinterpretation [2].

bug6 exposes incorrect exception flag generation in *ARA*'s FP conversion module for integer-to-float cast instructions. When an integer value exceeds the destination format's representable range, IEEE 754 requires the OF flag to be set while NV must remain clear [1]. In *ARA* the two flags are swapped due to an erroneous gate in the flag assignment logic of the shared cast unit in the FPU. Although the defect exists in shared code affecting all integer-to-float widths, it only manifests for FP16 destinations. FP32 and FP64 can represent (although potentially inexact) the entire range of any 64-bit integer, so overflow never occurs and the swapped flags are never triggered. FP16's limited range overflows with even modest inputs, exposing the bug. The presence of OF and NV flag discrepancies with the destination register agreeing between the golden reference and *ARA* confirms that only flag generation is affected.

bug7 corresponds to a missing NaN-boxing check in *ARA*. When a FP register holds a narrow-format value (e.g., FP32) in a wider register file (e.g., 64-bit), the RISC-V specification requires the unused upper bits to be all ones, which is known as NaN-boxing [2]. The div/sqrt module in *ARA* fails to validate this condition before consuming operands.

TABLE I: Reported failures for different *test setups*

Test Setup			Isolated Failing Instructions																									
ID	DUT	TS	fcvt.h.l	fcvt.h.lu	fcvt.h.w	fcvt.h.wu	fcvt.wu.d	fcvt.wu.s	fdiv.d	fdiv.h	fdiv.s	fld	flh	flw	fmax.d	fmax.s	fmin.d	fmin.s	fmul.d	fmv.x.h	fnmsub.d	fsd	fsh	fsw	fsqrt.d	fsqrt.h	fsqrt.s
1	*Spike SF*	pos	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	*Spike SF*	neg	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	*Spike FF*	pos	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1	-	-	-	-	-	-	-
4	*Spike FF*	neg	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	*VP++ SF*	pos	-	-	-	-	-	-	-	-	-	-	-	-	95	108	107	108	-	-	-	-	-	-	-	-	-	-
6	*VP++ SF*	neg	-	-	-	-	-	-	-	-	-	344	125	418	27	20	19	15	-	23	-	352	179	398	-	-	-	-
7	*VP++ FF*	pos	-	-	-	-	368	324	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	*VP++ FF*	neg	-	-	-	-	69	84	-	-	-	343	125	418	-	-	-	-	-	22	-	352	138	397	-	-	-	-
9	*QEMU*	pos	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	*QEMU*	neg	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	*ARA*	pos	-	-	-	-	-	-	6	-	44	-	-	-	-	-	-	-	-	-	-	-	-	-	23	-	266	
12	*ARA*	neg	3	3	2	1	-	-	4	5	212	-	-	-	-	-	-	-	-	-	-	-	-	-	5	16	63	

TABLE II: Bugs identified using *Automated Failure Characterization*

bug-ID	DUT	Affected Instructions	Diff	Attributes	Bug
bug1	*VP++ SF*	`fmax.d/s`, `fmin.d/s`	float destination reg.	NaN	NaN handling
bug2a	*VP++ SF/FF*	`fld/h/w`	float destination reg., exception	-	FP enable check
bug2b	*VP++ SF/FF*	`fsd/h/w`	memory, exception	-	
bug3	*VP++ FF*	`fcvt.wu.d/s`	int. destination reg.	precision mismatch	NaN handling
bug4	*VP++ FF*	`fdiv/fmul/fnmsub.d`	OF flag	OF flag	OF flag generation
bug5	*VP++ SF/FF*	`fmv.x.h`	int. destination reg.	precision mismatch	precision-mismatch handling
bug6	*ARA*	`fcvt.h.l/lu/w/wu`	OF flag	OF/NV/NX flag	OF & NV flag generation
bug7	*ARA*	`fdiv.s/d`, `fsqrt.s/d`	FP destination reg.	NaN, precision mismatch	NaN-boxed check is missing
bug8	*ARA*	`fdiv.s/d`	DZ/NV flag	DZ/NV flag	incorrect behavior for $0.0/0.0$
bug9	*ARA*	`fdiv.s/d`, `fsqrt.s/d`	FP destination reg.	NX flag	one-ULP result error

Instead of treating an invalid boxed value as a canonical quiet NaN, the hardware interprets the raw bit pattern directly, producing an incorrect result in the destination register. The bug is consistently flagged by NaN and precision-mismatch attributes.

bug8 exposes incorrect exception flag generation in *ARA* when both operands are zero ($0.0/0.0$). According to the IEEE 754 standard, this operation constitutes an invalid operation and must return a canonical quiet NaN with only the NV flag set; the DZ must remain clear, as it is reserved exclusively for nonzero dividends divided by zero [1]. The bug manifests independently of bug7 and is confirmed by native FP64 test cases where both operands are genuine zero. The bug is additionally triggered as a secondary symptom of bug7: when invalidly NaN-boxed FP32 operands are passed through without validation, they are misinterpreted as genuine zeros and fall into the $0.0/0.0$ path, producing the same spurious DZ flag. Both the standalone FP64 case and the NaN-box-induced case are consistently detected through the spurious DZ flag alongside the correctly expected NV flag.

bug9 captures two failure patterns in *ARA*'s div/sqrt rounding path, both manifesting as a one-*Unit in the Last Place* (ULP) error in the FP destination register with the NX flag set. The first pattern is a missing rounding mode: the round-to-nearest, ties-to-max-magnitude mode (`rmm`) is absent from the rounding case statement and falls through to the default truncation behavior, causing the result to be one ULP too low whenever `rmm` is selected explicitly or via the dynamic rounding mode (`dyn`). The second pattern is an incorrect sticky bit computation for FP64 operands: the guard and sticky bit extraction range overlap by one bit position, causing `rup` rounding mode to silently truncate instead of rounding up on certain inexact results.

Overall, the evaluation confirms that *FP-RVVTS* achieves near-complete F/D functional coverage in our coverage model and exposes non-trivial bugs across simulators, FP-library integrations, and silicon-proven RTL, while *Dependency-Based Test-Case Minimization* and *Automated Failure Characterization* accelerate root-cause analysis.

V. Conclusions

This paper presented *FP-RVVTS*, a Sail-guided verification flow for RISC-V FP implementations that uses *Sail RISC-V* as executable architectural specification instead of a simulator-specific oracle. The flow combines grammar-based test generation, differential execution, failure isolation, dependency-based minimization, and automated failure characterization.

We evaluated *FP-RVVTS* on six DUTs spanning ISSs, FP-library integrations, and the silicon-proven PULP ARA RTL implementation. The generated test sets achieve near-complete F/D functional coverage and expose nine non-trivial bugs, four previously reported simulator bugs confirmed against *Sail RISC-V* and five newly discovered issues.

Overall, the results show that a Sail-guided oracle, combined with compact failure reduction and structured failure characterization, enables practical RISC-V FP verification from simulators to RTL. The complete *FP-RVVTS* framework, together with the presented extensions, versioned test sets, and evaluation artifacts, is available in reproducible open-source form.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria, and the ODE4EC-DIG project funded by Chips JU (Grant Agreement No. 101252715) and co-funded by the Austrian Research Promotion Agency (FFG).

REFERENCES

[1] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std. IEEE Std 754-2019, 2019.

[2] *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture, Version 20260120*, RISC-V International, 2026.

[3] *The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 20260120*, RISC-V International, 2026.

[4] A. Joannou, P. Rugg, J. Woodruff, F. A. Fuchs, M. van der Maas, M. Naylor, M. Roe, R. N. M. Watson, P. G. Neumann, and S. W. Moore, "Randomized testing of RISC-V CPUs using direct instruction injection," *IEEE Design & Test*, vol. 41, no. 1, pp. 40–49, 2024.

[5] "RISC-V sail model," https://github.com/riscv/sail-riscv.

[6] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," in *Symposium on Principles of Programming Languages (POPL 2019)*, 2019.

[7] M. Schlägl and D. Große, "Single instruction isolation for RISC-V vector test failures," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024, pp. 156:1–156:9.

[8] M. Schlägl, J. Reichhardt, and D. Große, "From generation to failure categorization: An open-source automated RTL verification framework for RVV," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2026, pp. 622–629.

[9] K. Ruep, M. Schlägl, and D. Große, "Late breaking results: Float fight – verifying floating-point behavior in RISC-V simulators," in *Design, Automation and Test in Europe Conference (DATE)*, 2026, pp. 1–3.

[10] J. R. Hauser, "Berkeley SoftFloat," https://github.com/ucb-bar/berkeley-softfloat-3.

[11] N. Zurstraßen, N. Bosbach, and R. Leupers, "FloppyFloat: An open source floating point library for instruction set simulators," in *Design, Automation and Test in Europe Conference (DATE)*, 2025, pp. 1–6.

[12] "RISC-V ISA simulator (spike)," https://github.com/riscv-software-src/riscv-isa-sim.

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

[14] M. Schlägl and D. Große, "Fast interpreter-based instruction set simulation for virtual prototypes," in *Design, Automation and Test in Europe Conference (DATE)*, 2025, pp. 1–7.

[15] "QEMU a generic and open source machine emulator and virtualizer," https://www.qemu.org, 2025.

[16] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020.

[17] C. Tain, S. Patil, and H. Al-Asaad, "Survey of verification of RISC-V processors," *Journal of Electronic Testing*, vol. 41, pp. 111–138, 2025.

[18] V. Herdt, D. Große, and R. Drechsler, "Towards specification and testing of RISC-V ISA compliance," in *Design, Automation and Test in Europe Conference (DATE)*, 2020, pp. 995–998.

[19] V. Herdt, S. Tempel, D. Große, and R. Drechsler, "Mutation-based compliance testing for RISC-V," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 55–60.

[20] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient cross-level testing for processor verification: A RISC-V case-study," in *Forum on Specification and Design Languages (FDL)*, 2020, pp. 1–7.

[21] N. Bruns, V. Herdt, E. Jentzsch, and R. Drechsler, "Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging," in *Design, Automation and Test in Europe Conference (DATE)*, 2022, pp. 1123–1126.

[22] L. Klemmer and D. Große, "EPEX: processor verification by equivalent program execution," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2021, pp. 33–38.

[23] F. Riese, V. Herdt, D. Große, and R. Drechsler, "Metamorphic testing for processor verification: A RISC-V case study at the instruction level," in *International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC)*, 2021, pp. 1–6.

[24] M. Chupilko, A. Kamkin, and A. S. Protsenko, "Open-source validation suite for RISC-V," in *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*, 2019, pp. 7–12.

[25] "RISCV-DV," https://github.com/google/riscv-dv, 2024.

[26] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *USENIX Security Symposium (USENIX)*, 2023, pp. 1307–1324.

[27] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *USENIX Security Symposium (USENIX)*, 2024, pp. 5341–5358.

[28] I. Diaz, J. Sans, J. Quiroga, L. Valente, M. Dominguez, M. Rodriguez, M. Moreto, N. Sonmez, O. Palomar, R. I. Genovese, V. Jimenez, and V. L. Guglielmi, "Functional verification of a RISC-V vector accelerator," *IEEE Design & Test*, vol. 40, no. 3, pp. 36–44, 2023.

[29] F. Kaiser, S. Kosnac, and U. Brüning, "Development of a RISC-V-conform fused multiply-add floating-point unit," *Supercomputing Frontiers and Innovations*, vol. 6, no. 2, pp. 47–56, 2019.

[30] T. Lu, A. Liu, B. Xia, and P. Liu, "Comprehensive RISC-V floating-point verification: Efficient coverage models and constraint-based test generation," in *Design, Automation and Test in Europe Conference (DATE)*, 2025, pp. 1–7.

[31] Imperas, "riscvovpsim," https://github.com/riscv-ovpsim/imperas-riscv-tests, 2025.