2025 Design, Automation & Test in Europe Conference (DATE 2025)

# Comprehensive RISC-V Floating-Point Verification: Efficient Coverage Models and Constraint-Based Test Generation

Tianyao Lu, Anlin Liu, Bingjie Xia, and Peng Liu*
*College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, 310027, China*
*Email: liupeng@zju.edu.cn

***Abstract*—The increasing complexity of processor architectures necessitates more rigorous functional verification. Floating-point operations, in particular, present significant challenges due to their extensive range of computational cases that require verification. This paper proposes a comprehensive approach for generating floating-point instruction sequences to enhance the verification of RISC-V. We introduce a constraint-based method for floating-point test generation and design efficient coverage models as input constraints for this process. The resulting representative floating-point tests are integrated with RISC-V instruction sequence generation through a memory-bound register update method. Experimental results demonstrate that our approach improves the functional coverage of RISC-V floating-point instruction sequences from 93.32% to 98.34%, while simultaneously reducing the number of required instructions by 66.67% compared to the Google RISCV-DV generator. Additionally, our method achieves more comprehensive coverage of floating-point types in instruction write-back data compared to RISCV-DV. Using the proposed approach, we successfully detect representative floating-point-related faults injected into the RISC-V processor CV32E40P, thereby demonstrating its effectiveness.**

***Index Terms*—RISC-V, Floating-Point, Processor Verification, Instruction Sequence Generation**

## I. Introduction

The rapid evolution of processor architectures has introduced unprecedented complexity, necessitating rigorous and comprehensive functional verification methodologies, as hardware errors can result in severe reliability and security threats [1]. Among the processor components, implementing the IEEE-compliant floating-point unit (FPU) presents unique challenges. The specialized format of floating-point data and the complexity of operations introduce intricate corner cases for FPU verification. Undetected FPU errors can result in severe reliability issues, as evidenced by incidents such as the Intel Pentium FDIV bug [2]. Consequently, floating-point verification has become an indispensable aspect of processor design.

Simulation-based verification is the predominant method for processor verification due to its scalability and versatility. It typically uses an Instruction Set Simulator (ISS) as a reference model. Test instructions are input into the ISS and the Design Under Test (DUT), and their execution traces are compared to identify errors. Verification depth depends on the quality and diversity of input instructions, necessitating the generation of comprehensive and high-quality instruction sequences. Extensive research on instruction sequence generation [3]–[6] often employs constrained random generation methods to cover corner cases verification engineers may not anticipate.

RISC-V [7] is an open-source Reduced Instruction Set Computing (RISC) architecture known for its modular design. Google's RISCV-DV [8] is the leading Instruction Sequence Generator for RISC-V. Implemented using SystemVerilog with the Universal Verification Methodology (UVM), RISCV-DV supports extended instruction sets and employs constrained random generation to produce assembly-format instruction sequences. This generator has been extensively used for the functional verification of RISC-V processors. However, its reliance solely on initializing floating-point registers with special values limits its ability to cover various floating-point data types for multiple operations, making it challenging to achieve adequate coverage for the floating-point extension instruction set.

To address the deficiencies in RISC-V instruction sequence generators for floating-point operations, this paper proposes a comprehensive approach for generating RISC-V floating-point instruction sequences. We introduce a constraint-based floating-point test generation method, coupled with efficient coverage models, to generate representative tests. These tests are integrated with RISC-V instruction sequence generation through a memory-bound register update method for comprehensive verification. Experimental results demonstrate the effectiveness of our proposed approach. The generated RISC-V floating-point instruction sequences provide higher functional coverage than those produced by RISCV-DV. We achieve more comprehensive coverage of floating-point types in instruction write-back data. Furthermore, using our approach, we detected representative floating-point-related bugs injected into the processor, demonstrating its effectiveness in identifying potential issues.

In summary, this paper makes the following contributions:

- We develop a constraint-based floating-point test generation method that incorporates efficient coverage models to produce representative floating-point tests.
- We integrate floating-point tests with instruction sequence generation using a memory-bound register update method, thereby enhancing the overall RISC-V instruction sequence generation process.
- Our approach improves the functional coverage of RISC-V floating-point instructions, provides more comprehensive coverage of floating-point types in instruction write-back data and boosts verification efficiency.

## II. Preliminaries and Related Work

### A. RISC-V's Floating-Point Extension

The floating-point extensions in RISC-V include single-precision (F), double-precision (D), and quad-precision (Q). For instance, the F extension introduces 32 floating-point registers (`f0-f31`) and a 32-bit floating-point control and status register (`fcsr`). The lower five bits (`fflags`) of `fcsr` are used to track floating-point exception flags, while three bits (`frm`) indicate the rounding mode. Instructions can utilize either a static rounding mode encoded in the instruction or a dynamic rounding mode indicated by the `fcsr`.

RISC-V adheres to the IEEE 754-2008 standard [9] for floating-point arithmetic. The RV32F extension comprises 26 instructions, categorized into arithmetic, conversion, comparison, classification, and load/store instructions. Arithmetic instructions include addition, subtraction, multiplication, division, square root, multiply-add related operations, and computations for maximum and minimum values.

### B. Related Work

Simulation-based verification is the predominant method for processor verification, with the quality of input instruction sequences being crucial for efficiency. Instruction sequence generation approaches can be broadly categorized into model-based and coverage-guided methods. Model-based approaches separate the processor architecture model from instruction generation, employing constrained random methods and translating constraints into Constraint Satisfaction Problems (CSP) [4] [5] or Satisfiability Modulo Theories (SMT) [6]. Enhancements include abstract CSP [10], decision tree algorithms [11], and execution path models [12]. Coverage-guided approaches focus on maximizing coverage through feedback mechanisms, employing bayesian networks [13], reinforcement learning [14], and fuzzing techniques [15].

For FPU verification, while formal methods such as equivalence checking [16], theorem proving [17], and symbolic simulation [18] offer mathematical correctness guarantees, their computational complexity often constrains their application to module-level verification due to hardware-to-mathematical model transformation requirements. Furthermore, the requisite re-modeling and re-verification processes for each design iteration significantly impede their scalability to processor-level systems. The efficacy of simulation-based verification, which remains the predominant approach, fundamentally relies on the quality of test generation. While specialized tools such as fpgen [19] and testfloat [20] have been developed, they face significant limitations: fpgen necessitates substantial effort in developing specialized solvers for arithmetic constraints, while testfloat's scope is restricted to random constraint generation of input operands for floating-point tests, lacking capabilities for constraining output operands or intermediate results.

For RISC-V, research on instruction sequence generation includes official test-suites [21], [22] offering hand-written sequences with limited coverage, and automated methods such as constrained random [8], constraint-based specification [23], coverage-guided aging [24], and direct instruction injection [25]. However, current generators for RISC-V floating-point extension instructions fail to thoroughly explore the characteristics of floating-point data, providing only limited coverage of data types by focusing on instruction-level randomness and a few special floating-point values. To address these limitations, we propose a comprehensive floating-point instruction sequence generation approach that enhances coverage and improves the verification of RISC-V processors.

## III. Comprehensive Floating-Point Instruction Sequence Generation

This section presents our approach for generating floating-point instruction sequences. We start with an overview, followed by detailed explanations of its relevant parts.

### A. Overview

Figure 1 presents an overview of our approach for generating floating-point instruction sequences. This approach consists of two primary elements: a floating-point test generator for creating the floating-point test database and an instruction sequence generator for producing instruction sequences and initializing floating-point data memory.

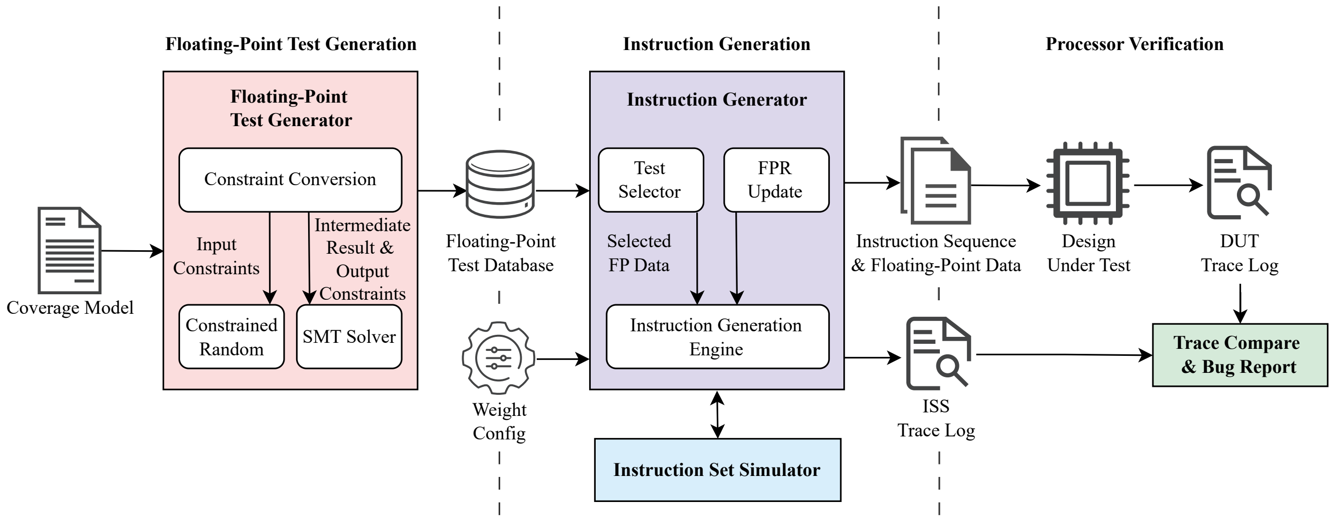
First, we develop a constraint-based floating-point test generator that utilizes an SMT solver and constrained random technique. This generator can parse floating-point operation constraints and generate the corresponding floating-point tests. We create a series of coverage models and input the relevant constraints into the test generator. The resulting representative floating-point tests form the floating-point test database.

Subsequently, we develop an instruction generator that integrates floating-point tests into the instruction generation process through a test selector and a memory-bound floating-point register (FPR) update method. This instruction generator employs an ISS for the dynamic simulation of instructions and the generation of ISS trace logs. The instruction generator produces the tuple containing the instruction address and machine code, as well as tuples for data addresses and floating-point data. These instructions and floating-point data can be directly injected into the memory of the processor under verification to generate the corresponding DUT trace log. Finally, we compare the DUT and ISS trace logs to identify potential bugs.

### B. Floating-Point Test Generation

In the floating-point test generation phase, we develop a constraint-based test generator specifically tailored to the unique encoding formats and arithmetic characteristics of floating-point numbers. For different types of floating-point verification test points, we create corresponding coverage models.

*1) Constraint-Based Floating-Point Test Generator:* For FPU test cases, the commonly used generation method involves constrained random generation for various floating-point arithmetic inputs [20]. However, due to the unique encoding format of floating-point data and the complexity of floating-point arithmetics, merely constraining the input operand is insufficient to cover all corner cases. Consequently, we propose a constraint-based floating-point test generation method that leverages the solving capabilities of the SMT solver to address



Floating-Point Test Generation
Instruction Generation
Processor Verification
Floating-Point Test Generator
Constraint Conversion
Input Constraints
Intermediate Result & Output Constraints
Constrained Random
SMT Solver
Coverage Model
Floating-Point Test Database
Weight Config
Instruction Generator
Test Selector
FPR Update
Selected FP Data
Instruction Generation Engine
Instruction Set Simulator
Instruction Sequence & Floating-Point Data
Design Under Test
DUT Trace Log
ISS Trace Log
Trace Compare & Bug Report

Fig. 1: Overview of comprehensive floating-point instruction sequence generation approach.

various constraints related to floating-point arithmetic. This method overcomes the limitations of previous approaches by not only enabling the constrained random generation of inputs but also utilizing the Z3 SMT solver [26] to handle constraints involving output operands and intermediate results. This approach allows for comprehensive constraint solving without the need to develop separate algorithms.

The encoding format of floating-point numbers consists of three parts: the sign bit $S$, the biased exponent field $E$, and the trailing significand field $T$. For 32-bit single-precision floating-point numbers, the lengths of these three parts are 1, 8, and 23 bits, respectively. Based on $E$ and $T$, floating-point numbers can be categorized into several basic types: normal numbers, subnormal numbers, and special values (infinity, zero, and NaN). Each operand can be constrained using three methods: basic types, ranges, and masks. These constraints can be combined and applied to various operands within the same arithmetic operation. To emphasize boundary conditions during verification, we incorporate specific boundary values into the basic type constraints of the test generator. The supported basic type constraints include ±Zero, ±MinSubnormal, ±Subnormal, ±MaxSubnormal, ±MinNormal, ±Normal, ±MaxNormal, ±Infinity, and NaN. To support more flexible test constraints, floating-point numbers can also be constrained using range constraints, such as [bf800000,00000001] (indicating values greater than or equal to -1 and less than or equal to +MinSubnormal). Additionally, mask constraints can be formed using “0”, “1” and “x”, where ”x” indicates that this bit can be either 0 or 1.

The floating-point arithmetic operands of interest include input operands, output operands, and intermediate results. For instance, in single-precision floating-point operations, both input and output operands are 32-bit floating-point numbers, while intermediate results vary depending on the specific implementation of the FPU. Each floating-point operation requiring rounding produces an intermediate result, which, after rounding, yields the final result conforming to the target format (i.e., the output operand). In certain implementations, the intermediate result can be represented by a floating-point number with a wider exponent field ($E$) and a wider trailing significand field ($T$). Due to variations in hardware implementations, our generator allows the specification of the $E$ and $T$ widths of the intermediate results using two parameters: ebits and sbits.

After designing constraint methods for the test generator, we perform constraint conversion for solving. The test generator categorizes constraints into two types: those on input operands only and those involving output operands and intermediate results. For input-only constraints, constraint random generation is feasible, while constraints involving outputs and intermediate results require an SMT solver due to their specific semantics.

*2) Coverage Model:* Floating-point arithmetic encompasses an extensive test space, necessitating the selection of representative subsets. To facilitate this selection, we develop efficient coverage models. Each model combines verification tasks with constraints input into our test generator. This generator then produces corresponding tests to build the test database. Our coverage models are primarily constructed from two aspects: floating-point data types and the characteristics of arithmetic.

The coverage model for floating-point data types partitions the test space at the data level into disjoint subsets. Using equivalence partitioning and boundary-value analysis, we categorize each floating-point operand into 17 distinct types: ±Zero, ±MinSubnormal, ±Subnormal, ±MaxSubnormal, ±MinNormal, ±Normal, ±MaxNormal, ±Infinity, and NaN. This coverage model targets each rounding mode for every floating-point arithmetic operation, combining all possible types for each operand into unique solving tasks. For instance, for floating-point addition with five rounding modes and three operands (two inputs and one output), there are $17^3 \times 5 = 26,565$ solving tasks. However, the actual number of solvable tasks is considerably lower. After solving each task in this coverage model using the test generator, the representative tests for solvable cases are stored in the test database.

Additionally, we develop a series of coverage models targeting the characteristics of floating-point arithmetic. These

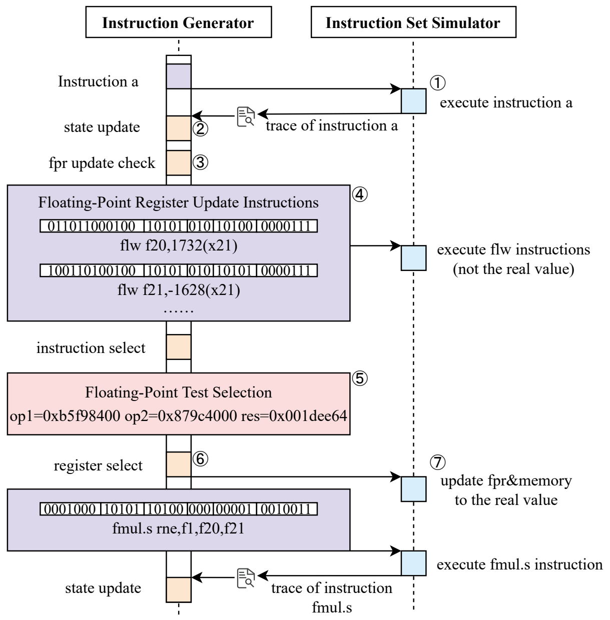


Fig. 2: Floating-point instruction generation flow.

models encompass boundary conditions for rounding, overflow, and underflow, as well as cancellation and shift in addition and multiply-add operations. Due to our test generator's support for intermediate result constraints, these coverage models can be translated into corresponding constraints for solving.

## C. Floating-Point Instruction Generation

During the instruction sequence generation phase, we integrate a floating-point test selector and a memory-bound register update method into our model-based instruction generator, guiding the generation of floating-point instruction sequences.

Figure 2 illustrates the process of generating floating-point instructions. Steps ①② represent the basic instruction generation approach. Step ③ checks whether the floating-point registers need updating. If an update is required, step ④ inserts a sequence of floating-point register update instructions. Step ⑤ utilizes the floating-point test selector to choose floating-point tests. Steps ⑥⑦ update the selected floating-point test data into the registers and memory bound by step ④, completing the memory-bound register update method.

*1) Instruction Generator:* Our basic instruction generator employs a model-based test generation approach to dynamically generate instruction sequences [27]. After generating each instruction, it executes using the ISS ①, which retrieves the corresponding instruction trace. This trace includes the values of registers, memory, and program counter before and after instruction execution. The generator uses the obtained trace to update its state information ②, ensuring the correct generation of subsequent instructions. This process continues until the required number of instructions is generated.

*2) Memory-Bound Register Update:* With the basic instruction generator established, we introduce a memory-bound register update method to efficiently integrate floating-point data into the instruction generation process. Typically, updating a floating-point register requires three instructions (`lui`, `addi`, and `fmv`) due to the absence of direct immediate value assignment instructions for floating-point registers. Our method decouples register update instructions from the selection of floating-point values, retrieving the value only when needed. This approach requires only one `flw` instruction to update a floating-point register value, reducing the number of register update instructions by two-thirds and enhancing the integration of floating-point tests into the instruction generation process.

After generating each instruction, an FPR update check ③ is conducted. As the number of floating-point registers used as source or destination registers increases, the probability of triggering the floating-point register update mechanism also rises. When this mechanism is triggered, a series of `flw` instructions ④ are inserted to update the floating-point registers. However, the initial values in memory corresponding to these `flw` instructions are not determined at this stage. During ISS simulation, the values in memory and the destination registers do not reflect the actual values. Each `flw` instruction's destination register and corresponding memory address are recorded in an FPR-memory binding list.

After generating the floating-point register update instructions, normal floating-point instruction generation continues. When an instruction $Ins$ with a floating-point register as the source register is generated, the test selector chooses the floating-point test ⑤. If specific source registers require a floating-point test, a register from the FPR-memory binding list is randomly selected as the source register ⑥. At this point, $Val$ is determined based on the floating-point test and serves both as the update value for the floating-point register $Reg$ and as the initial value stored at its bound memory address $Addr$, ensuring consistency between the register and memory states. The pair $(Addr, Val)$ is stored in the output tuple of data address and floating-point value, which subsequently forms part of the generator output file used to initialise the DUT memory, ensuring the correct execution of DUT. The generator then updates the internal register and memory state information. The ISS register and memory values are also directly modified to reflect the actual value $Val$ ⑦. The binding between the floating-point register and memory is then released, and the register $Reg$ is marked as used. When the ISS executes instruction $Ins$, the correct trace information is obtained. This process effectively integrates floating-point tests with instruction generation using the memory-bound register update method.

*3) Test Selector:* The test selector is designed to choose floating-point tests based on user-defined weights. There are two categories of floating-point tests. The first category is independent of instruction semantics, where special floating-point values, such as basic types and boundary values, are randomly selected for certain floating-point source registers. The second category is dependent on instruction semantics, in which the floating-point test is chosen by weight from the floating-point test database. The database is constructed based on coverage models described in Section III-B, considering the instruction's floating-point operation type and rounding mode.

Moreover, the test selector is not activated every time. Its activation probability can be configured to ensure that not every instruction uses updated registers. This approach helps maintain coverage of register dependencies between instructions.

*4) FCSR Refresh:* In RISC-V, the exception flags of floating-point operations, recorded in the `fflags` field of the `fcsr` register, accumulate until they are reset by software. To verify the correct raising and propagation of these exception flags, we design an optional FCSR refresh mechanism. This mechanism resets the `fflags` value using the `csrrw` instruction whenever an update to `fflags` is detected in the trace, following the generation and execution of each floating-point instruction.

## IV. Evaluation

This section evaluates the effectiveness of our RISC-V floating-point instruction sequence generation approach by addressing the following research questions:

- RQ1: Does our approach improve the functional coverage of floating-point instructions compared to existing methods? What factors contribute to this improvement?
- RQ2: In what ways do the floating-point tests generated based on our coverage models and the proposed memory-bound register update method enhance the quality of instruction generation?
- RQ3: How effective is our method in identifying bugs in the processor?

### *A. Experimental Setup*

We implement our test and instruction generators in C++ and modify QEMU [28] to support step-by-step interactive simulation of input instruction machine code, thereby establishing our co-simulation framework. We utilize the 32-bit RISC-V core CV32E40P [29], which supports the RV32IM[F|Zfinx]C instruction set, as the DUT. Our approach generates the 24 instructions in the RV32F extended instruction set in addition to the load/store instructions. We employ the industry-standard commercial tool Synopsys VCS to simulate the Register-Transfer Level (RTL) design. All experiments are conducted on an Intel® Core™ i7-10700 CPU with 32 GB of memory.

We compare our approach with the state-of-the-art RISC-V random instruction generator, RISCV-DV. For floating-point instructions, RISCV-DV includes the RISC-V floating-point arithmetic test, which can randomly generate 24 RV32F instructions, excluding load/store instructions.

### *B. Instruction Quality*

To demonstrate the effectiveness of our approach in generating high-quality floating-point instructions, we utilize the covergroup from RISCV-DV to collect functional coverage for the 24 single-precision floating-point instructions. This covergroup includes categories such as register coverage, data sign coverage, floating-point value coverage, and hazard coverage.

Figure 3 illustrates the functional coverage results for our full approach, our basic generator without the register update method, and RISCV-DV as the number of generated instructions increases. Our approach achieves a maximum coverage of 98.34%, outperforming RISCV-DV at 93.32% and the basic generator at 94.71%. Moreover, our approach reaches RISCV-DV's maximum coverage with only 2500 instructions, reducing the instruction count by 66.67%. This highlights our approach's superior efficiency and coverage.

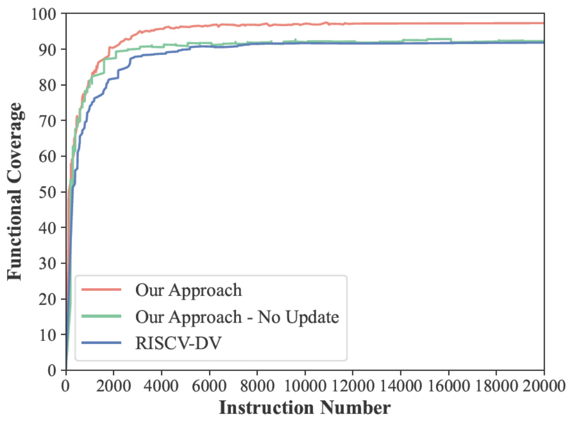


Fig. 3: Functional coverage comparison.

The basic generator's performance shows the importance of the register update method. While it surpasses RISCV-DV, its inability to handle corner cases, especially those involving operand diversity, prevents it from achieving the same level of coverage as our full approach.

The key advantage of our approach lies in enhancing the coverage of floating-point values across instructions. The diversity of generated data supporting this improvement is detailed in Section IV-C.

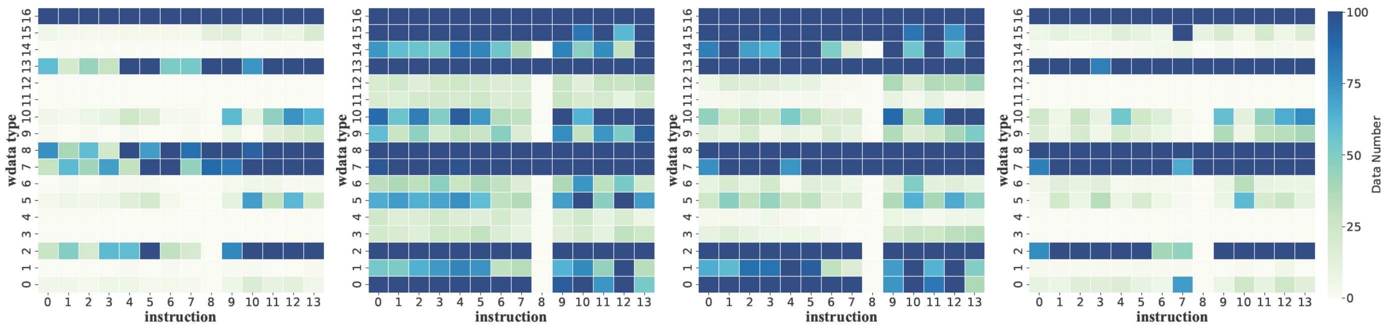
### *C. Data Diversity*

To illustrate the diversity of the instructions generated by our approach in terms of floating-point data, we record the write-back data for instructions where both the source and destination registers are floating-point registers. As shown in Figure 4, we document 5000 valid write-back data in each experiment and present them as heatmaps.

Figures 4a and 4b compare the diversity of floating-point data generated by RISCV-DV and our approach. Our approach successfully covers all possible data types for each instruction. In contrast, RISCV-DV lacks a specialized guidance mechanism for floating-point data, as it initializes floating-point registers only at the beginning of instruction generation. Consequently, many data types remain uncovered, with NaN being the most frequent write-back data type. This occurs because, in floating-point arithmetic, if an input operand is NaN, the output operand will also be NaN, leading to NaN propagation. Our approach resolves this issue through the memory-bound register update method, resulting in a more uniform data distribution.

### *D. Component Analysis*

We conduct a series of ablation experiments to demonstrate the effectiveness of our approach. Figures 4b, 4c, and 4d present the results. Figure 4c disables the second category of floating-point tests in Section III-C3, meaning it does not utilize the semantically related database generated based on the coverage models. Figure 4d disables the register update method, initializing floating-point registers only with various types of floating-point values.



(a) RISCV-DV (b) Our Approach (c) Our Approach - No Database (d) Our Approach - No Update

Fig. 4: Data diversity heatmap. The horizontal axis of the heatmap corresponds to instructions whose source and destination registers are both floating-point registers: [fmadd.s, fmsub.s, fnmsub.s, fnmadd.s, fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s, fsgnj.s, fsgnjn.s, fsgnjx.s, fmin.s, fmax.s]. The vertical axis represents various floating-point types of the write-back data, specifically [-Infinity, -MaxNormal, -Normal, -MinNormal, -MaxSubnormal, -Subnormal, -MinSubnormal, -Zero, +Zero, +MinSubnormal, +Subnormal, +MaxSubnormal, +MinNormal, +Normal, +MaxNormal, +Infinity, NaN].

The experimental results indicate that, without the register update method, the diversity of generated instruction data is comparable to that of RISCV-DV, leaving many floating-point types uncovered. When the register update method is applied but the database is not utilized, a greater variety of floating-point data is covered; however, many boundary values remain unaddressed. By incorporating the database, we achieve comprehensive coverage of all possible floating-point data types. This demonstrates the effectiveness of both our proposed register update method and the floating-point tests generated based on the coverage models.

## E. Performance Evaluation with Injected Bugs

To evaluate the effectiveness of our approach in processor verification, we employ a bug injection strategy. We inject 5 representative floating-point-related bugs, each covering common functional errors that may occur in the RTL core.

**B0:** When decoding the `fsub.s` instruction, the floating-point addition data path is mistakenly used without inverting the sign bit of the value in the second source register, causing the subtraction operation to be performed as an addition.

**B1:** A STUCK-at-0 fault is injected at the lowest result bit of the `fdiv.s` instruction.

**B2:** The rounding mode of the floating-point fused multiply-add (FMA) arithmetic is fixed to RNE (Round to Nearest, ties to Even).

**B3:** In the floating-point FMA arithmetic, the sticky bit of the third input operand is incorrectly calculated after being shifted, as the lowest bit is ignored. This can lead to the miscalculation of the intermediate result.

**B4:** In floating-point FMA operations, the overflow exception flag is incorrectly not raised in scenarios where overflow occurs before rounding.

Table I presents the detection results of the injected bugs from our approach and RISCV-DV. The "Result" column indicates whether each bug is detected, while the "#Aver. Exec. Instr." column shows the average number of instructions executed to detect these bugs over 10 runs.

TABLE I: Inject Bug Results

| | Our Method | | RISCV-DV | |
|---|---|---|---|---|
| Bug | Result | # Aver. Exec. Instr. | Result | # Aver. Exec. Instr. |
| B0 | ✓ | 49 | ✓ | 326 |
| B1 | ✓ | 33 | ✓ | 462 |
| B2 | ✓ | 44 | ✓ | 238 |
| B3 | ✓ | 5,453 | ✓ | 26,797 |
| B4 | ✓ | 125 | ✗ | – |

Experimental results demonstrate that our instruction generation approach effectively detects floating-point-related bugs injected into the RTL core. Compared to RISCV-DV, our approach identifies these bugs more efficiently with fewer instructions. Moreover, through our implementation of the FCSR refresh method and coverage models targeting the characteristics of floating-point arithmetic, we are able to detect floating-point exception flag-related bugs that RISCV-DV fails to identify during floating-point instruction sequence generation.

## V. Conclusion

This paper presents a comprehensive approach for generating floating-point extension instruction sequences for RISC-V, significantly improving functional verification of floating-point operations. By combining constraint-based test generation with a memory-bound register update strategy, we improve instruction diversity and coverage. Our experiments show increased functional coverage with fewer instructions and greater diversity in write-back floating-point data. Additionally, our method effectively detects floating-point-related bugs, demonstrating its utility in processor verification. These findings highlight the potential of our approach to improve processor functional verification, particularly for floating-point operations, paving the way for more reliable RISC-V implementations.

## Acknowledgment

This work was supported in part by the National Natural Science Foundation of China Key Program (Grant No. 92464205), in part by HEXIN Technologies Co., Ltd. Program, and in part by Zhejiang Province Key R&D Program (Grant No. 2025C01193).

## References

[1] Y. Wang, P. Liu, W. Wang, X. Wang, and Y. Jiang, "On a consistency testing model and strategy for revealing RISC processor's dark instructions and vulnerabilities," *IEEE Transactions on Computers*, vol. 71, no. 7, pp. 1586–1597, 2022.

[2] A. Edelman, "The mathematics of the Pentium division bug," *SIAM review*, vol. 39, no. 1, pp. 54–67, 1997.

[3] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a constraint satisfaction formulation and solution techniques for random test program generation," *IBM Systems Journal*, vol. 41, no. 3, pp. 386–402, 2002.

[4] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: Innovations in test program generation for functional processor verification," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 84–93, 2004.

[5] P. Liu, W. Hu, D. Liu, X. Han, and Y. Liu, "A RISC-V test sequences generation method based on instruction generation constraints," *Journal of Electronics and Information Technology*, vol. 45, no. 9, pp. 3141–3149, 2023.

[6] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using SMT-solver state generation," *Science of Computer Programming*, vol. 118, pp. 60–76, 2016.

[7] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume I," 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[8] "RISC-V DV." [Online]. Available: https://github.com/google/riscv-dv

[9] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[10] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract CSP," in *2012 Design, Automation & Test in Europe Conference & Exhibition*, 2012, pp. 15–20.

[11] Y. Katz, M. Rimon, A. Ziv, and G. Shaked, "Learning microarchitectural behaviors to improve stimuli generation quality," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 848–853.

[12] M. Chupilko, A. Kamkin, A. Kotsynyak, and A. Tatarnikov, "MicroTESK: specification-based tool for constructing test program generators," in *13th International Haifa Verification Conference*, 2017, pp. 217–220.

[13] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th Design Automation Conference*, 2003, pp. 286–291.

[14] N. Pfeifer, B. V. Zimpel, G. A. Andrade, and L. C. dos Santos, "A reinforcement learning approach to directed test generation for shared memory verification," in *2020 Design, Automation & Test in Europe Conference & Exhibition*, 2020, pp. 538–543.

[15] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *2019 Design, Automation & Test in Europe Conference & Exhibition*, 2019, pp. 360–365.

[16] A. Kapoor, W. Ferguson, H. Jain, and S. Kundu, "Formal verification of floating-point division," in *2023 IEEE 30th Symposium on Computer Arithmetic*, 2023, pp. 93–96.

[17] D. M. Russinoff, *Formal verification of floating-point hardware design*. Springer, 2022.

[18] J. Kleinekathöfer, A. Mahzoon, and R. Drechsler, "Polynomial formal verification of floating point adders," in *2023 Design, Automation & Test in Europe Conference & Exhibition*, 2023, pp. 1–2.

[19] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "FPgen-a test generation framework for datapath floating-point verification," in *Eighth IEEE International High-level Design Validation and Test Workshop*, 2003, pp. 17–22.

[20] "Berkeley testfloat." [Online]. Available: https://www.jhauser.us/arithmetic/TestFloat.html

[21] "RISC-V ISA TESTS." [Online]. Available: https://github.com/riscv-software-src/riscv-tests

[22] "RISC-V Architecture Test SIG." [Online]. Available: https://github.com/riscv-non-isa/riscv-arch-test

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

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

[25] 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.

[26] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[27] A. Liu, T. Lu, Y. Xi, Y. Liu, and P. Liu, "Enhancing functional verification with dynamic instruction generation by exploiting processor runtime states," in *2024 IEEE International Test Conference*, 2024, pp. 36–40.

[28] F. Bellard, "QEMU, a fast and portable dynamic translator," in *FREENIX Track: USENIX Annual Technical Conference*, 2005, pp. 41–46.

[29] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.