Constrained Random Verification for RISC-V: Overview, Evaluation and Discussion*

Sallar Ahmadi-Pour[1], Vladimir Herdt[1,2], Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{sallar,vherdt,drechsler}@uni-bremen.de

Abstract

RISC-V is a modern open and free *Instruction Set Architecture* (ISA) that is designed in a very modular way and enables to integrate custom instruction extensions in order to build highly application specific solutions. Extensive verification and validation is crucial to ensure that the design meets all requirements from the specification. *Constrained Random Verification* (CRV) has been shown to be a very effective technique for this purpose. *RISC-V DV* is a powerful CRV framework that is tailored for RISC-V and under active development by Google. In this paper we provide an overview, evaluation and discussion of CRV for RISC-V, based on the RISC-V DV framework. In our evaluation we assess the bug hunting capabilities of RISC-V DV by means of mutation samples and we provide additional execution metrics for the framework. Moreover, we add a discussion on the approach and sketch ideas for future research directions in this area to further boost the approach.

1 Introduction

RISC-V [31, 32] is a royalty free open source *Instruction Set Architecture* (ISA) that had a significant impact on recent advancements in the design of embedded systems, for example in the *Internet-of-Things* (IoT) context. RISC-V has extensive potential to become a game changer in this area and as of today is being strongly adopted in academia and industry. Beside being open and free, a key feature of RISC-V is the modern clean slate design that puts a strong emphasis on a high modularity and extensibility. It is possible to choose between different architecture bitwidths as well as standard instruction set extensions, such as multiplication and division, or atomic operations. Further customization is achieved by integrating custom instruction set extensions, a use-case that is specifically considered by the design of RISC-V. This allows to build highly application specific solutions that work very efficiently in combination with limited resources.

Extensive verification and validation is very important to ensure that the system fulfills the specification requirements with regard to functional as well as extra-functional aspects. Due to their ease of use and scalability, simulation-based methods still form the backbone to drive the verification and validation effort. In this context, *Virtual Prototypes* (VPs) are commonly employed as simulation backends [16]. A central component of the VP is the *Instruction Set Simulator* (ISS), which is an abstract model of the processor and thus responsible to process instructions one after another. Simulation-based methods require strong test generation techniques to achieve comprehensive results. *Constrained Random Generation* (CRG) [34, 26] is a powerful and scalable test generation technique that has been shown very effective in many different application areas, including functional verification [29, 14], validation of power management functionality [21], or error resiliency evaluations [33, 12] among others. CRG is a central building block of a *Constrained Random Verification* (CRV) framework. It works by specifying and solving constraints to generate new tests in a structured approach thus yielding better results than pure randomized generation. Coverage information is tracked and utilized to measure the verification progress as well as assess the verification quality.

RISC-V DV [4, 2] is a powerful CRV framework that is actively developed by Google and is tailored for RISC-V. It provides a large set of features including support for the complete 32 and 64 bit standard instruction extension set. RISC-V DV leverages SystemVerilog and UVM (*Universal Verification Methodology*) to generate tests in assembly format based on randomized instruction streams. Each test is executed on a reference system and the system under test (high-level simulator or RTL core). In a final step, the output of both systems is compared. This enables a comprehensive end-to-end verification and validation flow. RISC-V DV is designed as configurable framework that enables to select specific test generation strategies and simulation backends. Both of which can be further extended to meet application specific demands.

In this paper we provide an overview, evaluation and discussion of CRV for RISC-V, based on the RISC-V DV framework[1]. In our evaluation we perform a mutation-based analysis to assess the bug hunting capabilities of

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001.

[1]Visit `http://www.systemc-verification.org/risc-v` to find our most recent RISC-V related approaches.

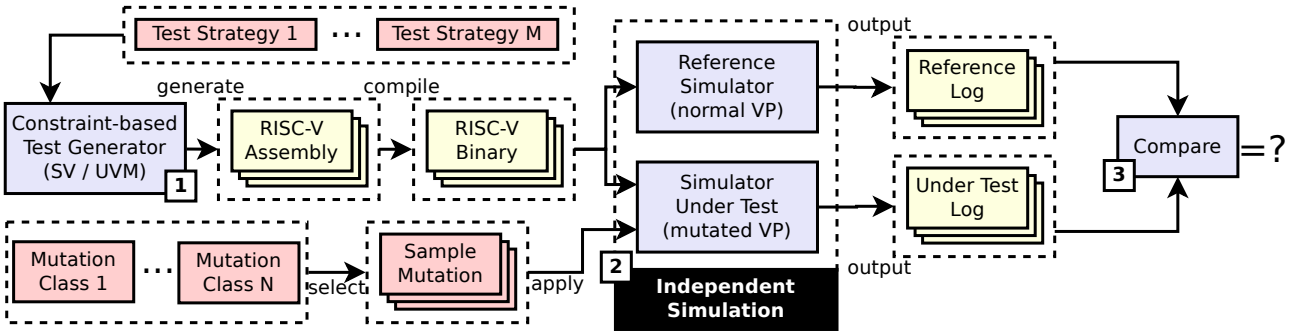

Figure 1 Overview on our evaluation setup to assess the bug hunting capabilities of the RISC-V DV framework. SV = SystemVerilog in the figure.

RISC-V DV. Therefore, we define mutation classes tailored for RISC-V and select elements from these classes for the evaluation. Each mutation represents a modification in the behavior of the simulator. In addition, we also provide execution metrics and statistics for RISC-V DV. Finally, we provide a discussion with a particular focus on future work with promising ideas to boost CRV for RISC-V further.

In the following, we start with an overview on related work (Section 2) and relevant background information on RISC-V (Section 3). Then, we describe our evaluation setup in more details and present our obtained results (Section 4). We end with the aforementioned discussion (Section 5) and then conclude the paper (Section 6).

2 Related Work

Several approaches have been proposed to generate processor-level stimuli for the purpose of verification. For example, they integrate model-based techniques with constraint solving [10, 25] or leverage coverage-guided test generation based on Bayesian networks [13] and other machine learning techniques [23] as well as fuzzing [28].

Recently, approaches specifically tailored for RISC-V verification have emerged. The baseline is provided by the official RISC-V unit and compliance tests [5, 3], which are directed test suites. The Scala-based RISC-V *Torture Test* [6] framework generates tests by integrating randomized instruction templates. Compared to RISC-V DV the feature set is much more limited. In [18], a cross-level verification methodology has been presented that uses a reference ISS in a tight co-simulation setting with an RTL core under test. A randomized instruction stream generator is integrated that evolves the instruction stream dynamically at runtime. While the technique is very generic and efficient, it requires a deep pipeline understanding of the RTL core under test to achieve the instruction stream integration in the co-simulation setting. Other RISC-V test generation approaches leverage constraint-based specification mechanism to define coverage requirements [17], utilize coverage-guided fuzzing techniques to generate tests [19, 15], and use symbolic execution techniques to find specific inputs [22].

In addition to test-generation methods, there are also a few formal verification approaches for RISC-V. Notable approaches that leverage model checking are *riscv-formal* [9], the *OneSpin 360 DV* RISC-V verification app [8] and the *C-S2QED* [11] approach for pipelined microarchitectures. However, formal methods may be susceptible to scalability issues in general and can benefit from complementary verification techniques to cross-validate them.

For the past decades the area of mutation-based testing has been investigated as a part of software testing techniques [24]. In [27] mutation testing is applied for VPs and embedded software systems. Additionally [22] show an approach for mutation-based compliance testing for RISC-V ISS. Through [30] mutation-based verification techniques have been proposed for the domain of RTL based hardware descriptions.

In this paper we provide an overview, evaluation and discussion of CRV for RISC-V, based on the RISC-V DV framework. We leverage mutation-based testing techniques inspired from the software domain to assess the test generation quality.

3 Background on RISC-V

As mentioned in the introduction, RISC-V is a royalty free and open source ISA that offers enourmous flexibility in building application specific solutions. RISC-V features an extremely modular design. The foundation is a mandatory base integer instruction set denoted RV32I, RV64I or RV128I with corresponding register widths. On top of that optional extensions, typically denoted as single letters, e.g. M (multiplication/division), A (atomics), C (compressed instructions) etc. are defined. Thus, RV32IC denotes a 32 bit ISA with the C extension. It has 32 *General Purpose Registers* (GPRs) each with 32 bit width. Compressed instructions have a length of 16 bit and are mapped to corresponding 32 bit uncompressed instructions from the base integer ISA. It provides different instruction classes such as arithmetic, load/store and branch/jump. They access registers (source: RS1 and RS2, destination: RD) and immediates to perform their operation. All instruction in RV32I have a length of 32 bit. Please refer to the RISC-V unprivileged ISA specification [31] for more information on RV32I and the available standard instruction set extensions, which includes a description of the instruction

Table 1 Chosen RISC-V DV test strategies and their description

Name	Description	# of instructions
`basic_arithmetic_test`	Arithmetic instruction test, no load/store/branch/jump instructions	10000
`rand_instr_test`	Random instruction stress test	10000
`jump_stress_test`	Stress back-to-back jump instruction test	5000
`loop_test`	Random instruction stress test with loops	10000
`unaligned_load_store_test`	Unaligned load/store test	6000

format and semantic. In addition, RISC-V provides *Control and Status Registers* (CSRs), which are special purpose registers that for example enable trap handling and environment interaction, which are described as part of the RISC-V privileged architecture specification [32].

4 Evaluation

In this section we present our mutation based testing approach that we use to evaluate the RISC-V DV framework. We start with an overview.

4.1 Overview

Fig. 1 shows an overview of our evaluation setup with the RISC-V DV framework. Following the RISC-V DV testing approach, the evaluation performs three subsequent main steps. In the first step, the SystemVerilog/UVM based test generator produces a set of tests in RISC-V assembly. Therefore, different test strategies can be used that control the test generation process according to the provided SystemVerilog constraints. In the second step the assembly tests are compiled into executable RISC-V binary files (ELFs) which are executed one after another on both, the normal VP (reference simulator) and the mutated VP (simulator under test). In Section 4.2 we present mutation classes from which we manually chose a set of specific sample mutations (for example replace a '+' operator of the *ADD* instruction with a '-' operator) for this evaluation. The mutated VP is obtained by applying the respective sample mutation in the ISS of the normal VP. A mutation is killed, in case a mismatch is detected in the execution between the normal and mutated VP. Therefore, each VP generates an execution trace, in the RISC-V DV trace format, for each executed test. The trace records changes of the internal simulation state and provides information on each executed RISC-V instruction. This includes each *program counter*, changes to a GPR or CSR, as well as a disassembly of the respective instruction. In the third and last step, both traces are compared to detect mismatches in the simulation results. Based on traces the RISC-V DV framework can also compute functional coverage information by using SystemVerilog covergroup definitions.

In the following, we first present our mutation classes in Section 4.2 that we consider for this evaluation. Then, we provide more details on the actual evaluation setup in Section 4.3. Next, we show and discuss our obtained results in Section 4.4. Finally, we present additional information on the instruction distribution of selected test strategies in Section 4.5.

4.2 Mutation Classes

We have defined six mutation classes for this evaluation that represent common faults that may occur during the implementation:

M1 Modify a single bit in the result of an instruction that is written to the destination register RD.

M2 Swap the operator of an instruction with another (e.g. change + to -, $\ll$ to $\gg$, etc.).

M3 Add or remove an unary operator (e.g. !, - and ~) to a calculation.

M4 Modify the behavior of the branch instructions (e.g. swap == with !=, <= with <, etc.).

M5 Change the alignment of bytes for load and store instructions, i.e. shift values, treat signed as unsigned and vice versa.

M6 Change the immediate format in an immediate instruction for another (e.g. change an I-type immediate for an B-type immediate)

We chose a set of specific sample mutations as representatives from these six mutation classes based on user experience.

4.3 Setup

For this evaluation we use our our open source RISC-V VP [7, 20]. We focus on the base 32 bit RISC-V ISA with compressed instruction extension support (RV32IC).

We consider 5 different test strategies which are provided by RISC-V DV and focus on arithmetic, branches and jumps, memory access, and randomized instructions[2]. **Table 1** shows a short description (column: Description) for each of the 5 test strategies (column: Name) and the configuration setting on the number of instructions to be generated (column: # of instructions)[3]. The values used in Table 1 are the default values for the test strategies by the RISC-V DV framework. We apply each test strategy one after another on every sample mutation (to see if it kills the mutation).

In total we chose 21 sample mutations and perform 10 iterations for each of the 5 test strategies. This makes a total

[2]RISC-V DV supports some additional specialized test strategies that focus on testing of CSRs, interrupts and the MMU. We have not included them, as they go beyond the scope of our RV32IC evaluation.

[3]This number of instructions is interpreted as a guideline by the test strategy and not as a fixed requirement, thus the number of instructions varies around that configuration value.

Table 2 Results of our evaluation using the 21 sample mutations with the 5 different test strategies. For each test strategy 10 iterations have been performed, i.e. 10 test cases generated by RISC-V DV.

Mutation Class	Instruction	(Sample) mutation description	# of tests that did not kill the mutation	# of tests that did kill the mutation	% of mutations killed by the tests	Average # of instructions until mismatch				
						basic_arithmetic_test	rand_instr_test	jump_stress_test	loop_test	unaligned_load_store_test
M1	ADD	Mask result bit 3 with constant 1	0	50	100	66.9	62.5	213.4	60.2	65.1
	SUB	Mask result bit 1 with constant 0	8	42	84	217.6	264.4	571.0	272.9	73.0
	AND	Mask result bit 5 with constant 1	9	41	82	717.2	109.6	370.0	128.0	71.0
	LUI	Mask result bit 0 with constant 1	0	50	100	5.0	5.0	5.0	5.0	5.0
M2	ADD	Swap + for -	6	44	88	130.3	112.4	484.1	118.0	72.3
	SUB	Swap - for +	8	42	84	99.7	463.6	463.6	162.7	73.0
	AND	Swap & for |	9	41	82	708.4	181.8	405.0	140.5	71.0
	ORI	Swap | for ^	11	39	78	1186.5	319.2	993.9	306.0	68.0
M3	XOR	Introduce ~at RS2	8	42	84	702.1	99.1	471.3	128.6	63.5
	OR	Introduce ~at RS2	9	41	82	716.0	124.2	410.2	102.3	68.0
	AND	Introduce ~at RS2	9	41	82	716.5	173.9	401.1	401.1	71.0
M4	BNE	Swap != for >	21	29	58	-	1049.9	903.0	714.0	-
	BLT	Swap < for <=	29	21	42	-	3062.4	1510.0	1041.2	-
	BGE	Swap >= for >	28	22	44	-	1912.9	1503.8	884.1	-
M5	SB	Shift content for store left by 8	39	11	22	-	565.2	-	-	76.0
	LB	Shift content for load left by 8	38	12	24	-	112.8	-	-	74.0
	LB	Treat loaded value as unsigned	38	12	24	-	120.9	-	-	74.0
	LBU	Treat loaded value as signed	37	13	26	-	223.1	-	-	64.7
M6	ANDI	Change I-Imm for J-Imm	9	41	82	717.1	276.8	652.5	192.6	65.0
	SLTI	Change I-Imm for J-Imm	33	17	34	3173.0	2129.4	2255.0	2001.4	-
	XORI	Change I-Imm for B-Imm	9	41	82	751.8	348.3	623.7	258.9	74.0
		Total	358	692	65.9					

of $21 \cdot 10 \cdot 5 = 1050$ tests that are executed on the normal and mutated VP.

All tests have been performed on an Intel Xeon Gold 6240 processor running Ubuntu 20.04.1 LTS and using the commercial RTL simulator QuestaSim 2019.4 with SystemVerilog/UVM support for test generation.

4.4 Results

Table 2 shows the results grouped by the chosen 18 sample mutations. The table is seperated by double columns into three horizontal parts:

1. The left part describes the respective sample mutation X. It shows the mutation class of X, the instruction opcode on which X is applied, and a short description of X.
2. The middle part provides information on how many of the 50 tests do kill the respective sample mutation.
3. The right part presents for each of the 5 test strategies how many instructions are executed on average (for all 10 test iterations) until the mutation is killed. The character '-' denotes that the mutation has not been killed by the respective test strategy.

From Table 2 it can be observed that the highest percentage of mutants were killed in the mutation class M1 (82% to 100%) followed by M2 (82% to 88%), M3 (82% to 84%), M4 (42% to 58%), M5 (22% to 26%) and M6 (34% to 82%). The test strategies are tailored for specific use-cases and thus are suitable to find different errors. The `basic_arithmetic_test` strategy did not kill mutants from the M4 and M5 classes since it does not focus on branch or jump instructions. Similarly, `jump_stress_test` and `loop_test` did not kill mutants from M5 and `unaligned_load_store_test` did not kill mutants from M4. Therefore, it is important to combine multiple test strategies to obtain comprehensive results.

The mutation in the LUI instruction has always been killed after 5 instructions in each test because a respective LUI instruction is used for initialization purposes in every test. In total 692 of the 1050 tests killed a mutation, which cor-

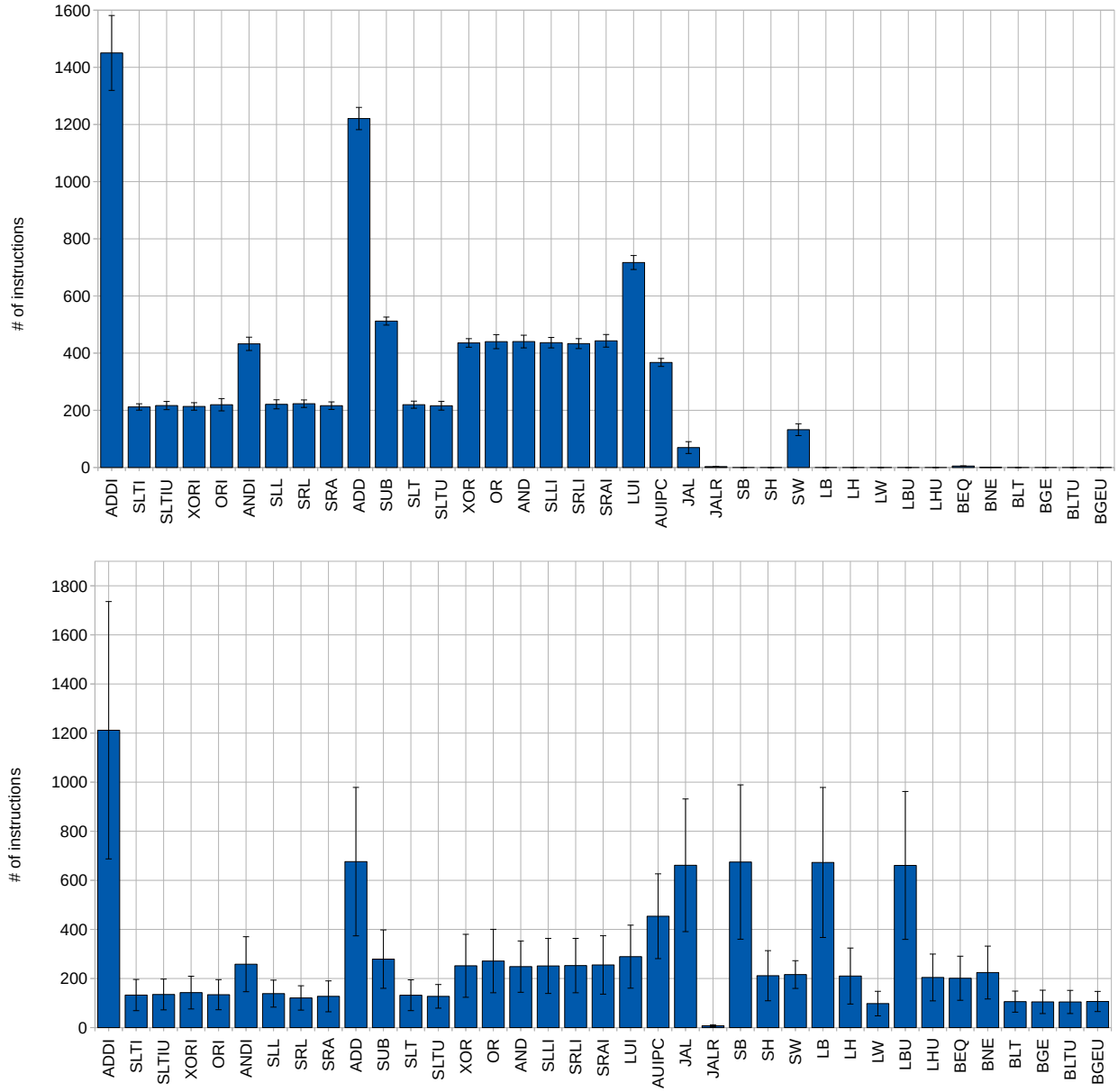

Figure 2 Distribution of instructions averaged over 10 generated tests. The tests have been generated by RISC-V DV using the `basic_arithmetic_test` (top histogram) and `random_instruction_test` (bottom histogram) test strategies, respectively.

responds to 65.9% of the tests. In combination the test strategies were able to kill all considered sample mutations at some point.

The average number of instructions until a mutant is killed is derived from the logfiles generated by the test framework. For each mutant the number is added up to an average value if the mutant was killed by the test. Thus, this value can help to find anomalous events as shown by the LUI instruction.

In total, around 8000 instructions are processed per second. This includes the test generation, execution on both VPs, and comparison of the trace logs.

4.5 Instruction Distribution

Fig. 2 exemplarily show the distributions of instructions for the `basic_arithmetic_test` and `random_instruction_test`. The error bars in the histograms represent the deviation that the average value has for that specific instruction. This way each average value also illustrates the spread of the 10 test iterations that were executed.

For both histograms it can be observed that the instructions ADDI and ADD occur more often than other instructions. These instructions are used to setup values in the registers for calculations.

The majority of the distribution in the `basic_arithmetic_test` (Fig. 2 top) is approximately even for the arithmetic instructions in the different instruction classes. Store, branch and jump instructions occur significantly less often, which is to be expected for an arithmetic test.

In the histogram of the `random_instruction_test` (Fig. 2 bottom), also a very even distribution can be ob-

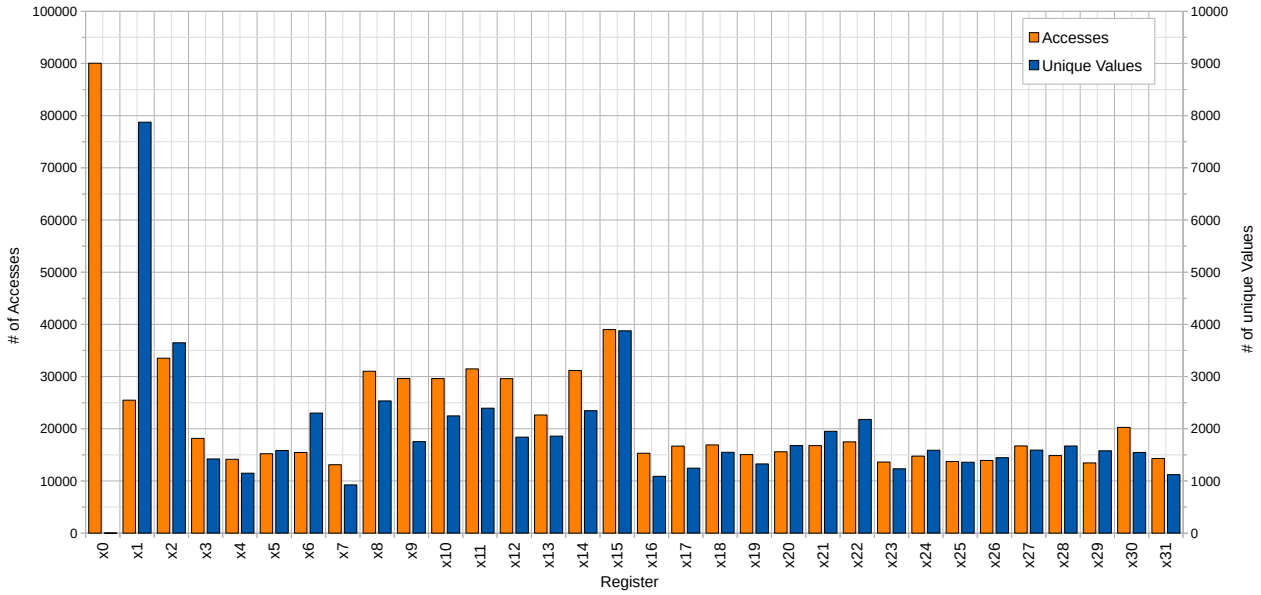

Figure 3 Distribution of the number of register accesses (orange bar, left each) and number of unique register values (blue bar, right each) over all tests. The primary y-axis (left) shows the number of accesses to a register, the secondary y-axis (right) shows the number of unique values written to a register.

served for the different instruction classes[4]. The main distribution is at around 200 instructions accross all instructions. A major outlier is the JALR instruction which performs a register-based jump and thus is difficult to test in a randomized way (since arbitrary jumps can cause runtime errors).

These histograms exemplarily indicate that the RISC-V DV test strategies provide a good distribution of instructions tailored for the respective strategy. This statement is supported by the other test strategies as well (we omitted their histograms for brevity).

4.6 Register and Immediate Coverage Statistics

We analyzed all binaries generated by the RISC-V DV framework to obtain coverage information with respect to register and immediate access statistics. The access statistics are combined for all observed RISC-V instructions.

Fig. 3 shows the distribution of the number of register accesses (orange bar, left each) and number of unique register values (blue bar, right each). Please note the different scaling of the two y-axes. Most registers were accessed between 10000 and 30000 times with register `x0` having around 90000 accesses. This difference with register `x0` can be explained with the fact, that `x0` is not a pure general purpose register but hardwired to zero. Thus, there is only a single unique value available in `x0`. It can also be observed that the relation between the number of accesses and number of unique values stored in the registers is mostly consistent.

Table 3 shows coverage information based on access statistics of immediate fields. This includes the different RISC-V immediate types (as specified by the RISC-V specification per instruction) including the *SHAMT* (shift amount) field. The first and second rows of the table show the number of accesses and observed unique values for the respective immediate field. The last row gives a percent value that denotes how many of the possible values of the respective immediate field have been covered. For example, *I-imm* is a 12 bit field with, thus has a possible value range of 4096 values. With the observed 3595 values (combined over all executed instructions with I-imm fields), 87.77% of the possible values have been covered. Based on this evaluation, it can be observed that in particular the coverage for the branch and jump immediates can be further improved.

[4]For example load and store byte (LB,LBU,SB), branch equal and not equal (BEQ,BNE), branches with comparison (BLT,BGE,BLTU,BGEU), immediate shifts (SLLI,SRLI,SRAI), etc.

5 Discussion and Future Work

RISC-V DV is a powerful CRV framework tailored for RISC-V with already strong bug hunting capabilities for the RISC-V base ISA, as indicated by our experimental evaluation. To boost CRV for RISC-V further, for future work we plan to:

- Provide a more extensive evaluation with additional mutation classes that are comprehensively processed together with a larger number of test iterations. In addition to using an ISS, also perform the evaluation on an RTL core and evaluate an ISS/RTL cross-level setting.
- Utilize RISC-V DV for different use-cases that go beyond functional verification. Advanced extra-functional use-cases include error resiliency evaluation, testing of information flow tracking or side channel evaluations. This requires designated test generation techniques and coverage metrics to be effective.

Table 3 Number of accesses and unique values in the observed immediate fields over all tests (combined for all observed instructions)

Description	SHAMT	I-Imm	S-Imm	B-Imm	J-Imm	U-Imm
# of accesses	49363	105932	15991	27006	24777	27209
# of unique values	32	3595	1928	212	99	1722
% of possible values covered	100%	87.77%	47.07%	5.18%	0.01%	0.16%

- Integrate coverage information in a feedback loop with the test generation process to guide the instruction stream generation towards maximizing the coverage goals faster. RISC-V DV leverages SystemVerilog/UVM constraints as foundation for the test generation, thus an efficient coverage-guided loop would require to dynamically evolve the constraint descriptions at runtime.
- Provide designated support for RISC-V instruction set extension with the RISC-V DV framework. This requires to generate appropriate constraint classes to enable test generation for the new instruction set. In addition, the new test generators need to be tightly coupled and interleaved with the existing ones to ensure a comprehensive and efficient test generation process. We envision to extract instruction set constraints for the purpose of test generation from a *Domain Specific Language* (DSL) for functional RISC-V ISA description (such as the CoreDSL [1]).
- Investigate an integration of dynamic information, available through the ISS simulation model, with the SystemVerilog/UVM constraint-based test generation process. We envision to guide the test generation process by accessing the architectural state of the ISS on-the-fly during execution. Based on this precise runtime information, instruction generated can be more efficiently and significantly simplified, for example by avoiding illegal memory accesses and infinite loops more easily by exactly knowing the current execution state of the ISS.
- Boost performance by employing parallelization and by utilizing FPGAs for test execution in particular when using an RTL core simulation (e.g. in a cross-level setting with a reference ISS).
- Investigate *Artificial Intelligence* (AI) methods to help in detecting suceptible regions for functional bugs and other issues inside of a simulator or RTL core in order to guide the test generation process. This requires an appropriate integration with the constrained random generator.

6 Conclusion

In this paper we provided an overview, evaluation and discussion of CRV for RISC-V, based on the RISC-V DV framework. RISC-V DV is a powerful CRV framework that is under active development by Google and provides strong constraint-driven test generation methods tailored for RISC-V. In our evaluation we assessed the bug hunting capabilities of RISC-V DV by means of mutation testing and provided relevant execution metrics. Our experiments indicate the RISC-V DV is effective in finding common implementation bugs. Finally, we provided an extensive discussion that identifies promising research directions for future work to further boost CRV for RISC-V.

7 References

[1] CoreDSL. `https://github.com/Minres/CoreDSL`.

[2] Open source RISC-V processor verification platform. `https://riscv.org/wp-content/uploads/2019/12/12.10-16.10b-Open-Source-Verification-Platform-for-RISC-V-Processors.pdf`.

[3] RISC-V compliance task group. `https://github.com/riscv/riscv-compliance`.

[4] RISC-V DV. `https://github.com/google/riscv-dv`.

[5] RISC-V ISA tests. `https://github.com/riscv/riscv-tests`.

[6] RISC-V torture test generator. `https://github.com/ucb-bar/riscv-torture`.

[7] RISC-V virtual prototype. `https://github.com/agra-uni-bremen/riscv-vp`.

[8] OneSpin 360 DV RISC-V Verification App. `https://www.onespin.com/solutions/risc-v`, 2020.

[9] RISC-V formal verification framework. `https://github.com/SymbioticEDA/riscv-formal`, 2020.

[10] B. Campbell and I. Stark. Randomised testing of a microprocessor model using SMT-solver state generation. In F. Lang and F. Flammini, editors, *Formal Methods for Industrial Critical Systems*, pages 185–199, 2014.

[11] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz. Gap-free processor verification by S2QED and property generation. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 526–531, 2020.

[12] S. El-Ashry, H. Ibrahim, M. A. Ibrahem, M. Khamis, A. Shalaby, M. AbdElsalam, and M. W. El-Kharashi. On error injection for noc platforms: A uvm-based practical case study. In *Proceedings of the 10th In-*

ternational Workshop on Network on Chip Architectures, NoCArc'17, New York, NY, USA, 2017. Association for Computing Machinery.

[13] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC*, pages 286–291, 2003.

[14] F. Haedicke, H. M. Le, D. Große, and R. Drechsler. Crave: An advanced constrained random verification environment for systemc. In *2012 International Symposium on System on Chip (SoC)*, pages 1–7, 2012.

[15] V. Herdt, D. Große, and R. Drechsler. Closing the RISC-V compliance gap: Looking from the negative testing side. In *Design Automation Conf.*, 2020.

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

[17] V. Herdt, D. Große, and R. Drechsler. Towards specification and testing of RISC-V ISA compliance. In *Design, Automation and Test in Europe*, pages 995–998, 2020.

[18] 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*, 2020.

[19] V. Herdt, D. Große, H. M. Le, and R. Drechsler. Verifying instruction set simulators using coverage-guided fuzzing. In *Design, Automation and Test in Europe*, pages 360–365, 2019.

[20] V. Herdt, D. Große, P. Pieper, and R. Drechsler. RISC-V based virtual prototype: An extensible and configurable platform for the system-level. *Journal of Systems Architecture - Embedded Software Design*, 2020.

[21] V. Herdt, H. M. Le, D. Große, and R. Drechsler. Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach. In *Forum on Specification and Design Languages*, pages 1–8, 2017.

[22] V. Herdt, S. Tempel, D. Große, and R. Drechsler. Mutation-based compliance testing for RISC-V. In *ASP Design Automation Conf.*, 2021.

[23] C. Ioannides, G. Barrett, and K. Eder. Feedback-based coverage directed test generation: An industrial evaluation. In S. Barner, I. Harris, D. Kroening, and O. Raz, editors, *Hardware and Software: Verification and Testing*, 2011.

[24] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

[25] Y. Katz, M. Rimon, and A. Ziv. Generating instruction streams using abstract CSP. In *DATE*, pages 15–20, 2012.

[26] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 258–265, 2007.

[27] C. Kuznik and W. Muller. Native binary mutation analysis for embedded software and virtual prototypes in systemc. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 290–291, Dec 2011.

[28] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *ISSTA*, pages 261–272, 2009.

[29] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek. Constraint-based random stimuli generation for hardware verification. In *Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence - Volume 2*, pages 1720–1727. AAAI Press, 2006.

[30] Y. Serrestou, V. Beroulle, and C. Robach. Functional verification of RTL designs driven by mutation testing metrics. In *DSD*, pages 222–227, 2007.

[31] A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. 2019.

[32] A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. 2019.

[33] D. You, I. Amundson, S. A. Hareland, and S. Rayadurgam. Practical aspects of building a constrained random test framework for safety-critical embedded systems. In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, MoSEMInA 2014, pages 17–25, New York, NY, USA, 2014. Association for Computing Machinery.

[34] J. Yuan, C. Pixley, and A. Aziz. *Constraint-based Verification*. Springer, 2006.