Efficient Cross-Level Processor Verification using Coverage-guided Fuzzing

Niklas Bruns
Institute of Computer Science, University of Bremen
Bremen, Germany
nbruns@uni-bremen.de

Vladimir Herdt
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
vherdt@uni-bremen.de

Daniel Große
Institute for Complex Systems, Johannes Kepler University
Linz, Austria
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
daniel.grosse@jku.at

Rolf Drechsler
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
drechsler@informatik.uni-bremen.de

ABSTRACT

In this paper, we propose a novel simulation-based cross-level approach for processor verification at the Register-Transfer Level (RTL). We leverage state-of-the-art coverage-guided fuzzing techniques from the software domain to generate processor-level input stimuli. An Instruction Set Simulator (ISS) is utilized as a reference model for the RTL processor under test in an efficient co-simulation setting. To further boost the fuzzing effectiveness, we devised custom mutation procedures tailored for the processor verification domain. Our experiments using the popular open-source RISC-V based VexRiscv processor demonstrate the effectiveness of our approach in finding intricate bugs at the processor level.

CCS CONCEPTS

• **Hardware → Simulation and emulation**; **Equivalence checking**.

KEYWORDS

Cross-Level Verification, Processor Verification, Fuzzing

ACM Reference Format:
Niklas Bruns, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2022. Efficient Cross-Level Processor Verification using Coverage-guided Fuzzing. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3526241.3530340

GLSVLSI '22, June 6–8, 2022, Irvine, CA, USA

ACM ISBN 978-1-4503-9322-5/22/06…$15.00
https://doi.org/10.1145/3526241.3530340

1 INTRODUCTION

The fast-growing *Internet-of-Things* (IoT) and wearable markets have unleashed a high demand for scalable and customized computing cores with rapidly changing requirements. As a reaction, *Instruction Set Architectures* (ISAs) are in demand like never before. In particular, the modern free and open source RISC-V [38, 39] ISA is designed in a modular and extensible way in order to facilitate building very application specific processors tailored for the specific use-case at hand. Thus, many emerging embedded systems integrate a RISC-V core at their heart. Highly efficient techniques for processor verification at the *Register-Transfer Level* (RTL) are crucial these days to keep up with the short innovation cycles. Due to their ease of use and scalability, simulation-based verification techniques are still prevalent. However, they rely on strong test generation techniques for processor-level input stimuli to ensure a thorough verification process. In the software domain, fuzzing is such a strong test generation technique that has been shown very effective, versatile and comparatively easy to use. With it's roots going back to [34], fuzzing since then enjoyed great popularity and celebrated enormous successes in various verification scenarios. State-of-the-art fuzzing techniques employ so called coverage-guided techniques that rely on mutation-based algorithms to generate new inputs. Notable representatives in this modern *Coverage-guided Fuzzing* (CGF) category are AFL [6] and LLVM libFuzzer [7]. Both have a very impressive bug finding record [6, 7]. Recognizing the fuzzing achievements, Microsoft and Google have launched large scale cloud-based fuzzing services [8, 9]. Moreover, fuzzing is used as continuous testing technique in many prominent software projects such as Chromium [3]. Despite the tremendous and ongoing success story of fuzzing in the software domain, the applications in the hardware domain are much more limited thus far.

Contribution: In this paper, we propose to leverage state-of-the-art CGF techniques for processor verification at the RTL. An *Instruction Set Simulator* (ISS) is utilized as a reference model for the RTL-core under test in an efficient cross-level co-simulation setting. The fuzzing process is guided by the code coverage of the reference ISS as well as by the core under test. Since the fuzzer can generate unstructured randomized input data, we carefully devised

our co-simulation to enable feeding the same instruction sequences to both models and supporting arbitrary instruction sequences, including the handling of potential infinite loops. To boost the fuzzing effectiveness, we developed custom mutation procedures tailored for the generation of common instruction patterns. As a case study, we present results on the verification of the popular open source RISC-V based VexRiscv processor [15]. Our experiments demonstrate that 1) our fuzzer optimizations improve the verification result statistically significant compared to a baseline state-of-the-art CGF, and 2) our fuzzing-based approach has been very effective in finding numerous intricate bugs in VexRiscv.

2 RELATED WORK

Simulation-based approaches that rely on test generation techniques have a long history in the processor verification domain, hence several approaches have been proposed to improve the generation of processor-level stimuli for verification purposes. One direction is to employ model-based test generators which use an input format specification to guide the generation process [17]. Constraints are leveraged for specification purposes and processed by *Constraint Satisfaction Problem* (CSP) / *Satisfiability Modulo Theories* (SMT) solver in [20, 21]. Optimization techniques to propagate constraints among multiple instructions more effectively have been presented in [29]. [32] proposed to mine processor manuals to obtain an input model automatically which can be utilized for test generation purposes. Other notable approaches include coverage-guided test generation based on bayesian networks [22] and other machine learning techniques [28]. Moreover, formal methods based on symbolic execution techniques have been utilized for test-case generation at the ISS level [37]. Finally, fuzzing-based techniques have been employed to test processor emulators by comparing their execution results against a physical CPU [33]. For RISC-V specifically, a number of verification techniques have emerged recently. The baseline are semi hand-written directed test-suites that cover different RISC-V instruction sets [2, 12, 18]. Other simulation-based approaches generate instruction sequences by combining predefined randomized patterns [1] and by utilized constraint-based specifications [14, 24]. CGF based on LLVM libFuzzer has been proposed for RISC-V ISS verification [26]. However, the approach works by generating standalone executable test-cases instead of using a co-simulation environment and targets a different level of abstraction than our approach. [27] also propose a cross-level co-simulation based approach for processor verification at the RTL which was extended by [19]. These approaches are arguably the most similar to our approach. However, these approaches has some limitations in it's usability and required manual effort. They generate one single endless instruction stream that dynamically evolves at runtime, this means that for the same program counter, different instructions will be returned over time. Moreover, in combination with the different fetch behavior of the ISS and RTL-core, for example the core has a pipeline with branch prediction and hence performs speculative pre-fetching of instructions, this dynamic instruction stream property makes the co-simulation setup very complex. A deep pipeline understanding is required to feed the same instructions into the ISS and RTL-core as well as extract register values at the right time points to compare execution results. In addition, the setup is highly architecture specific and thus requires significant manual effort to support different configurations. Finally, a custom instruction stream generator is employed to generate the single endless instruction stream, which again requires significant effort to setup effectively. In contrast, our proposed approach enables a much more simplified co-simulation setup by generating test-cases one after another. This also allows to test different configurations of the core much more easily. At the same time, we also do not impose restrictions on the test-cases, i.e. our setup supports arbitrary control flows (including self-loops) and load / store instructions. Moreover, we rely on existing state-of-the-art fuzzing algorithms which we further boosted with lightweight domain specific fuzzing extensions. In combination we achieve an easy to use, yet very powerful framework for comprehensive processor verification at the RTL. Only very few approaches using fuzzing for hardware verification have been proposed. [31] combines fuzzing with FPGA acceleration. [35] presents a fuzzer for SpinalHDL designs and automates the generation of input corpus and fuzzer harness. Another approach has been presented in [36]. At first glance, the approach seems similar to ours, but there are some fundamental differences. The goal of the approach is to verify individual IP blocks such as AES or HMAC, i.e. HW peripherals, rather than verifying an entire processor core as in our approach. Also, they generate TileLink bus protocol instructions for their testbench implementation, while our approach generates processor core instructions directly. This has the advantage that the fuzzer does not have to follow a bus protocol, and as a consequence, it does not require a bus-centric grammar but can robustly mutate binary instructions. Additionally, their approach does not include a method to check for errors in the DUT automatically. Lastly, in the RISC-V domain, a few formal approaches have been proposed which are based on model checking techniques [11, 13], but formal techniques may be susceptible to scalability issues.

3 PRELIMINARIES

3.1 AFL

American Fuzzy Lop (AFL) [6] is an out-of-process coverage-guided grey box fuzzer. Out-of-process fuzzers, in contrast to in-process fuzzers, reset the whole process and the *Software Under Test* (SUT) does not require a custom reset function. AFL uses its trim mutation to reduce the size of each test vector (without changing the measured coverage) because the execution of small test vectors trends to be faster than the execution of big test vectors. To discover new behaviors, AFL uses a manifold of mutations. The detection of new behaviors is realized through edge coverage. Notable mutations are the bitflip mutations, the arithmetic mutations and havoc mutation. The bitflip mutation flips a variety number of bits, the arithmetic mutation adds/subtracts integers and the havoc mutation is a combination of a multitude of individual mutations and applies them at random positions.

3.2 RISC-V

RISC-V is a free and open ISA that is defined by the global non-profit organization RISC-V International [10]. A major goal during the definition of RISC-V was to create a suitable ISA for nearly any computing device. Hence, RISC-V has a very modular design.

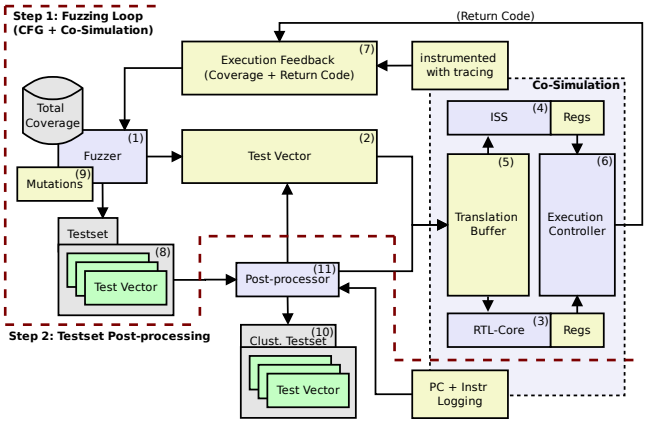

Figure 1: Overview on processor verification

The ISA specification consists of two volumes[38, 39]. The first volume is called the *Unprivileged Specification* and the second one *Privileged Specification*. RISC-V consists of multiple n-bit-base integer sets (RV32I, RV64I, RV128I) and several optional standard base extensions like multiply/divide (M) or compressed instructions (C). Moreover, space for custom instructions has been reserved. These base integer sets and extensions are defined in the first volume. The second volume specifies the RISC-V privileged architecture. It includes important functionalities required for operating systems and other sophisticated hardware / software interactions. Central aspects for these functionalities are the *Control and Status Registers* (CSRs), which are registers serving a special purpose.

4 PROCESSOR VERIFICATION USING FUZZING

In this section, we present our cross-level processor verification approach that is based on co-simulation and *Coverage-Guided Fuzzing* (CGF). Fig. 1 shows an overview. Essentially, our approach consists of two subsequent steps: 1) a fuzzing loop based on CGF (top of Fig. 1) to generate a set of test vectors, and 2) postprocessing (below dashed line in Fig. 1) to reduce the generated set. In the following, we provide a general overview on both steps and then detail the most important parts in the following subsections. The complete flow starts with the CGF-based fuzzing loop. The Fuzzer (1) generates test vectors (2) which are used as instruction stream for the co-simulation. The co-simulation is a combination of the RTL-core (3) under test and a reference ISS (4). An essential component of the co-simulation is the *Translation Buffer* (5) which transforms a fuzzer generated (bounded) test vector into an endless instruction stream through cyclic repetition. The Translation Buffer helps in decoupling the co-simulation setup from micro-architecture specific optimizations such as pipelining. We will present more details on the Translation Buffer in Section 4.1. The RTL-core and ISS execute the (endless) instruction stream delivered by the Translation Buffer. We use a maximum instruction execution count in the ISS in combination with heuristic cycle detection techniques to efficiently set a runtime limit on each test vector execution. Parallel to the execution, the *Execution Controller* (6) of our approach checks whether the behavior of the processors are equal. This is realized through register value comparison. If the register values are not equal, the execution will be terminated with an error. The functional principle of the Execution Controller is described in Section 4.2. The whole co-simulation, which includes the RTL-core and ISS, is instrumented to collect coverage. The vast advantage of collecting the whole co-simulation coverage is that the coverage of one core acts as virtual coverage[1] for the other one. The coverage and the return code are given as execution feedback (7) to the fuzzer, in our approach using a shared memory. The fuzzer collects the test vectors (8) and categorize them in two sets. The first set contains all test vectors that result in an equal behavior for both processors and the second set contains all test vectors that are triggering a behavior mismatch. The fuzzer quits if a given fuzzing timeout is reached. To improve the verification results, we designed custom mutations (9) that enhance the fuzzing efficiency. As we have shown in our experiments, the enhancement is statistically significant (see Section 5). We provide a detailed description of this mutations in Section 4.3. To reduce the manual labor of verification engineers, we introduced a post-processing step (bottom of Fig. 1) that clusters the test vectors which trigger mismatches (10) in order to encapsulate test vectors that detect the same bug. Therefore, the co-simulation is compiled with more extensive logging instrumentation to provide the additional required feedback for the post-processing. The functionality of this post-processing (11) component is presented in Section 4.4.

Table 1: Translation Buffer execution example

Translation Buffer		ISS		RTL	
ID	Instr.	ADDR	Instr	ADDR	Instr
0	LWU x8, x0, 48	0x00	ID: 0	0x00	ID: 0
1	lb x6, x2, 52	0x20	ID: 1		
2	lw x6, x8, 49	0x24	ID: 2		
3	c.slli x6, 9	0x28	ID: 3		
4	c.addi4spn x14, 332			0x04	ID: 4
0	LWU x8, x0, 48			0x08	ID: 0
...					

4.1 Translation Buffer

In this section we describe our Translation Buffer that transforms the fuzzing test vector into a deterministic endless instruction stream. If the transformation would not be deterministic, the fuzzing performance would be greatly reduced because the fuzzer assumes a deterministic execution. Our Translation Buffer is based on the concept of a ring buffer. Generally, a ring buffer is a circular queue that works according to the FIFO principle. It is possible to write an infinite number of values into a ring buffer because if it is full, the oldest values will be overwritten. A typical ring buffer is empty when all values were read once. As you can see, a typical ring buffer solves the inverse of our actual problem, because we need infinite reading and not writing. Thus we need to customize the ring buffer. For our approach, the size of our Translation Buffer must be initialized with the number of instructions contained in the test vector with the consequence that no instruction has to be overwritten. When all instructions in the ring buffer are read, the internal read pointer

[1]Virtual coverage describes the concept of improving the coverage measurement granularity through inserting synthetic coverage points to enhance coverage-guided verification performance. A detailed explanation, as well as the advantages of virtual coverage, can be found in [23].

will be reset, and thus the Translation Buffer delivers the same instruction sequence as directly after the initialization. Thus, our Translation Buffer provides an endless instruction stream by cyclic repetition. In the following, we describe how our Translation Buffer is used to generate an endless instruction stream for the processor verification. Therefore, we prepared an RV32I example, shown in Table 1, that demonstrate the instruction generation in combination with the different fetch behaviors of the ISS and the RTL-core, caused by pipelining. The first third of the table shows the functionality of the Translation Buffer, the second third the use of the Translation Buffer values by the ISS, and the last third the use by the RTL-core. In this example, the fuzzer generates a 160bit test vector that is loaded into the Translation Buffer. Every entry of the Translation Buffer has the size of a 32bit instruction. Consequently, the Translation Buffer has five entries ($160 \div 32 = 5$). At the beginning, the ISS loads 0x00 as first instruction, because the PC is initialized with zero. The Translation Buffer supplies the LWU instruction at ID 0 to the ISS. Because LWU is a pure RV64I instruction, and hence invalid in RV32I, the ISS raises an illegal instruction exception and jumps to the trap handler, which has been initialized by default to the address 0x20. The ISS executes the instructions at 0x20 and 0x24 (ID 1 & 2) successfully. Then, the ISS loads the compressed instruction *c.slli x6, 9* at 0x28 (ID 3). The ISS raises an illegal instruction exception because compressed instructions are disabled in this setting. The ISS executes the trap loop until the Execution Controller terminates the ISS. After termination of the ISS, the Execution Controller starts execution of the RTL-core. The execution begins again at 0x00. To ensure that the ISS and the RTL-core get the same instruction stream, already-fetched instructions are cached in the Translation Buffer. In this example, the RTL-core illegally executes the LWU instruction successfully, and afterwards it executes the instruction at 0x04. This address was *not fetched* previously by the ISS, and as a consequence, a new value must be supplied by the Translation Buffer (ID 4). Because the RTL-core in our example has a multistage pipeline, it fetches the command at the address 0x08 at the same time as it tries to execute the command at 0x04. The Translation Buffer delivers the LWU instruction (ID 0) *again* because this Translation Buffer does not hold any further unused instructions. Now that we have described our Translation Buffers in detail, we will continue with the description of the Execution Controller.

4.2 Execution Controller

The purpose of the Execution Controller is twofold. Firstly, it prevents infinite loops, and secondly it detects mismatches between the processor cores. Because our test vectors are interpreted as arbitrary endless instruction streams, a test vector can result in an infinite loop. Due to the fact that the fuzzing performance hugely depends on the fast execution of test vectors, it is important to detect infinite loops as soon as possible. An infinite loop is detected (conservatively) when a new program counter address is equal to an already executed program counter address and the register values are unchanged too. Additionally, we set a hard limit of 10,000 instruction executions for the ISS to handle the halting problem. Now to the second purpose of the Execution Controller. As mentioned earlier, the second purpose is to detect processor core mismatches. It is challenging to compare very different implementations of the same functionality. This is especially true when comparing an ISS and a RTL-core with pipelining. In order to compare the functionality of the processor cores it is essential to identify important behaviors and to choose the right time to compare. We have observed that important functionalities sooner or later lead to a value change in a register. For example the instruction *add x1, x2, x3* reads the values of register *x2* and *x3*, adds them together and writes the result to x1. As a consequence, we compare the register values to detect functional mismatches. It is important that the register values are compared (or logged) right after the processor cores executed the respective instruction completely. Thus, a processor core instruction execution synchronization is needed because otherwise, many false mismatches are detected. It is easy to detect the complete instruction execution of an ISS but it is very difficult to detect in case of a pipelined RTL processor. A pipelined processor core speeds up the execution through parallelization and has no general signal to indicate a complete instruction execution. Moreover, comparing the registers too frequently leads to performance degradation. It is unnecessary to compare the registers when no value has changed. For example, if the addition instruction from before has not the target register x1 but x0 (*add x0, x2, x3*), then no register value changes because the value of register x0 is always zero. Thus, a comparison of the register values should only be executed if a register was changed. Consequently, we use for synchronization and comparison time points when register values were really changed. To demonstrate the behavior of the Execution Controller, we look again at Table 1. First, we look at the instruction execution of the ISS. The ISS raises an illegal instruction exception because *LWU x8, x0, 48* is no RV32I instruction and jumps to the trap handler at address 0x20. Then, it executes the LB and LW instructions which respectively change the register x6. The ISS does not execute the compressed instruction *c.slli* but raises an illegal instruction exception and jumps to the trap handler again. After the execution of the ISS, the RTL-core starts the instruction execution. In contrast to the ISS, the RTL-core, in our example, erroneously executes the *LWU x8, x0, 48* instruction (not *lb x6, x2, 52*) that leads to a change in the register *x8*. The Execution Controller detects a register value change and executes a register value comparison between the registers of the ISS and the RTL-core, discovers the mismatches between x8 and x6, throws an error and stops the simulation. Thus, a mismatch was found between the ISS and RTL-core.

4.3 Enhanced Mutations

To enhance the fuzzing performance of AFL, a state-of-the-art coverage-guided fuzzer, we designed a set of problem-specific mutations. The first is named *Fast Exploration* and the second *Enhanced Havoc*. In the following paragraphs, we will present these mutations.

4.3.1 Fast Exploration. The *Fast Exploration* is a deterministic mutation that was designed to boost the exploration speed of our fuzzer. To accomplish this, we add a preliminary exploration phase before the normal mutation procedure. It begins with the insertion of *each* RISC-V instruction at the beginning of *every* test vector. The value of each instruction argument is fixed to src/dest register x0 and immediate 0. For example, we insert the *addi x0, x0, 0*

instruction. After the instruction insertion, the fuzzer executes the newly generated test vector and saves it if it increases the coverage. By storing only test vectors that increase test coverage, you limit the state-space and thus prevent a state-space explosion. Next, we use the bitflip mutation. The purpose of the bitflips is to cover possible arguments and to uncover unknown instructions. These two mutations are iteratively repeated until no new test vectors are found. Thus, with this new mutation prephase, one can cover an extensive range of the RISC-V instruction sequence state space without encountering scalability problems or depending on a lucky random seed. Furthermore, this prephase has, for several reasons, a very low overhead. The first reason for the low overhead is that RV32I only contains 40 different instructions. The second reason is that the two operations in this phase are not applied to every generated test vector but only to test vectors that reach new coverage points. The third reason is that the bitflip mutation does not add any new actual overhead since bitflip was only moved to this phase and would otherwise have been executed later. For a detailed description of the AFL mutation flow, we refer to [5]. Implementing this mutation for our case study was straightforward because AFL's simplistic design makes adjustments to the control flow easy.

4.3.2 Enhanced Havoc. The original havoc mutation is a combination of single mutations applied at random positions. Similar to our Fast Exploration Mutation, we have added the insertion of RISC-V instructions. However, in contrast, the instruction arguments are not fixed to zero and also support compressed instructions. In addition to the insertion that makes the test vector longer, we added a replacement variant that does not change the size of the test vector. Furthermore, we have integrated improvements for CSR testing. As we mentioned earlier, the CSRs are the backbone of RISC-V privileged architecture [39]. To accommodate this, we have also added CSR instruction insertion/replacement functionality. Here, the functionality always adds two CSR instructions. The first instruction always writes a CSR, and the second reads the same CSR. Thus, a possible CSR misbehavior is propagated directly into the register and thus made detectable for the Execution Controller.

4.4 Post Processing

Fuzzing is a very efficient verification methodology. It generates many test vectors, reaches high coverage, and uncovers numerous bugs. After the test generation, it is crucial to investigate the reported errors carefully. During this procedure, it can often be noticed that many test vectors reveal the same bug. To save manual analysis time, it is helpful to cluster the test vectors that detect the same bug. In this section, we describe our automatic post-processing step for test vector clustering. Every cluster is represented through a unique test vector that behaves like every other test vector in this cluster. For the post-processing we use a custom version of the co-simulation which logs all executed instructions with the corresponding addresses. To use this version for fuzzing is not reasonable because it is much slower due to the hard disk write accesses. Another difference is, that the post-processing co-simulation does not need the coverage instrumentation which is essential for fuzzing. Next, we extract the instruction, which leads to the bug. The post-processing distinguishes the mismatches in two cases. Firstly, instruction addresses of the ISS and the RTL-core do not have deviations. In this case, the mismatch arises from a result difference of the last executed instruction. In the second case, an instruction address mismatch has occurred and leads to the case that different instructions have been executed. Thus, the erroneous instruction is the last instruction that was executed *before* the instruction address mismatch. The post-processor now clusters the test vectors based on the executed commands up to the point where the faulty command was executed.

Table 2: Fuzzing Results

	Vanilla AFL		Enhanced AFL	
Run	#Queue	#Unique-Crash	#Queue	#Unique-Crash
0	3312	217	2628	274
1	3008	223	2450	286
2	2439	206	2367	281
3	3264	237	2691	354
4	2219	163	2442	230
5	2505	263	2915	315
6	2385	310	2540	281
7	2335	207	3614	382
8	2551	276	2885	289
9	3313	244	2698	287
10	3127	265	2514	250
Mean	2768.90	237.36	2613.09	274.43
Median	2551.00	223.00	2614.00	281.00
Sum	30458.00	1619.00	28744.00	2021.00

5 EVALUATION

In this section, we present our case study and discuss the evaluation results. Our case study aims to evaluate the applicability of fuzzing in combination with co-simulation for processor verification. As *Device Under Test* (DUT), we choose the popular open source RISC-V VexRiscv processor, which is available as an RTL description. As reference ISS, we extracted the ISS from the open source RISC-V VP [25]. VexRiscv is a configurable and 4-stage pipelined RTL-core written in SpinalHDL, which is a higher-level language for VHDL/Verilog generation[4]. As a case-study, in the following, we use the RV32IM configuration of VexRiscv. However, compared to [27], which requires significant manual effort to setup an appropriate co-simulation for different processor configurations, our approach could also be directly used to test different processor configurations. RISC-V VP is a virtual prototype written in SystemC TLM that supports many RISC-V instruction sets. To enable the co-simulation, we translated the RTL-core to C++ using the free and open tool verilator [16] and embedded it into a common SystemC testbench with the ISS. Because SystemC has no functionality to reset the whole simulation inclusive the scheduler, we used as baseline the out-of-process fuzzer AFL 2.56b [6]. Our evaluation process is guided by [30] in order to guarantee quality and comparability. Due to the random nature of fuzzing, we used eleven runs per fuzzer and the standard statistical test from the fuzzing community named Mann-Whitney U Test. The Mann-Whitney U Test checks whether the central tendencies of two independent samples are different. It is a non-parametric test that makes no assumption about normal distribution and consequently works for small sample sizes[30]. To make the case study realistic, we used a run time limit of 24 hours. We use random seeds and as corpus, we considered the 32bit long

value $0x0000$ which essentially represents an empty corpus. All experiments are conducted on a Linux machine with an Intel Xeon Gold 5122 CPU with 3.60GHz. In the following, we compare the fuzzing results of AFL with the results of our (mutation) enhanced AFL version (Section 5.1). Afterwards, we present the bugs which we have found with our methodology in more details (Section 5.2).

5.1 Vanilla AFL vs. Enhanced AFL

Table 2 illustrates the fuzzer run results of our case study. It allows comparisons between the runs of the unmodified state-of-the-art fuzzer AFL 2.56b (*Vanilla AFL*) and our optimized AFL version (*Enhanced AFL*). For clarification, the values in the column *#Queue* are the number of of the test vectors that increase the coverage and cause no execution mismatch of the co-simulation, and the values in the column *#Unique-Crash* are the number of unique test vectors that cause an execution mismatch. The procedure for the identification of unique crashes has been described in Section 4.4. On average, Enhanced AFL generates fewer *#Queue* test vectors than Vanilla AFL, which is better as it can provide the same coverage results with fewer test vectors. We statistically analyzed the *#Queue* values with the one-tailed Mann-Whitney U Test. The critical threshold of the U-value at the confidence interval of 95% is 34. The U-value for the *#Queue* column is 60 (z-score: 0, p-value: 0.5). Therefore, even though the result appears to be better, the improvements are not statistically significant according to this standard statistical test. In practice, this means that this divergence is negligible. Also, it should be noted that the average line and branch coverage for Vanilla AFL and Enhanced AFL are equal. Thus, we can assume that the #Queue test vectors are of equivalent quality. On average, Enhanced AFL generates more *#Unique-Crash* test vectors than Vanilla AFL. Again, we statistically analyzed the values with the one-tailed Mann-Whitney U Test. The critical threshold of the U-value at the confidence interval of 99% is 25. The U-value is 17 (z-score: -2.8236, p-value: 0.0024). Because the p-value is lower than 0.1, the result is not only significant but actually *highly significant*. Thus, we have shown that Enhanced AFL is highly significant better at detecting errors. In total, the Unique-Crash test vectors of Vanilla AFL execute 20,4130 and of Enhanced AFL 135,139 instructions. As mentioned earlier, Enhanced AFL has generated more test vectors that successfully uncover bugs. Thus, Enhanced AFL generates more and shorter (less executed instructions) test vectors than Vanilla AFL . Shorter instruction sequences benefit debugging because the verification engineer has to analyze less state space and instruction behavior. Fig. 2 shows the instruction execution frequency of the unique test vectors. Overall, one can see in this graph that the test vectors of Enhanced AFL have a more uniform execution frequency than the test vectors of Vanilla AFL. The shorter test vectors and more uniform instruction execution frequency are a consequence of Fast Exploration because it injects many instruction & argument combinations at the beginning of the test vectors. Thus, we have shown that Enhanced AFL is highly significant better at detecting errors, it generates more test vectors with additionally better quality. In addition, we have also demonstrated that Enhanced AFL has no disadvantageous effects on overall test generation and coverage.

5.2 Manual Analysis

In our evaluation, we configured the VexRiscv to support the instruction subset RV32IM. We found some errors associated with the decoder and many associated with CSRs. In this section, we present the found errors. We will begin with the decoder bugs.

5.2.1 Decoder Bugs. The most obvious bug is that the VexRiscv executes the RV64I LWU instruction, i.e. the instruction is only valid on a 64 bit and not 32 bit RISC-V core. The execution should not happen because we configured the core only to support RV32IM instructions. Moreover, VexRiscv executes several illegal instructions. This behavior traces back to 3 similar decoding bugs of the *SLLI*, *SRLI* and *SRAI* instructions. These three errors have in common, that the encodings contain an additional incorrect don't care bit. Thus, the processor misinterprets many illegal instructions as SLLI, SRLI, or SRAI instruction, respectively.

5.2.2 CSR Bugs. Now we describe the CSR bugs. The RISC-V privileged architecture specifies the following four ID CSRs: *mvendorid*, *marchid*, *mimpid*, and *mhartid*. These CSRs are read-only and provide hardware identifiers. In these four CSRs, we found the same bug. VexRiscv erroneously does not raise an exception at a write attempt. The RISC-V privileged architecture also specifies many read-only counter CSRs for performance measurements. The following cycle CSRs have the same bug: *cycle*, *cycleh*, *instret*, *instreth*. A write attempt does not result in a raised exception. At the same time, all counter CSRs should be defined and allow a read access. However, VexRiscv erroneously raises an exception on a read access to the following counter CSRs: *time*, *timeh*, *hpmcounter*, *hpmcounterh*, *mcounteren*, *mhpmcounter3-31*, *mhpmcounter3-31h*, and *mhpmevent*. It should be noted that we also observed many value discrepancies for the counter CSRs. These mismatches are no bugs but the result of different micro-architectural details varieties. We also observed bugs in trap handling CSRs. If a trap is taken in machine mode, it is specified that the virtual address of the triggering instruction must be written into the *mepc* CSR. The implementation of this CSR is faulty in VexRiscv because the lowest two bits of the *mepc* CSR are not masked. The CSR *mtval* is quite similar to *mepc*. If an exception is raised, mtval is set to zero or to an exception-specific information. If an illegal instruction exception is raised, it is defined that the triggering instruction must be written into the CSR *mtval*. If we make an *ecall* with incorrect parameters, the VexRiscv core writes the instruction to *mtval*. This is a bug because it is not an illegal instruction, so it must write the virtual address of the instruction into *mtval*. In addition, we observed a mismatch between VexRiscv and the RISC-V VP ISS in case of compressed instructions. If we execute a compressed instruction with VexRiscv, the core correctly raises an illegal instruction fault and then fills the 16bit compressed instruction into the CSR mtval. In contrast, the RISC-V VP fills the whole decoded 32bit into the *mtval* instruction because the VP RV32IM decoder does not support compressed instruction (RV32C). However, the RISC-V specification is unclear at this point. This demonstrates that the specification itself needs clarification at specific points. Finally, we found a bug in the *misa* CSR implementation of VexRiscv. This CSR shows which instruction sets are enabled and allows to enable and disable instruction sets. We found out that the *misa* CSR of the VexRiscv cores allows arbitrary values like zero.

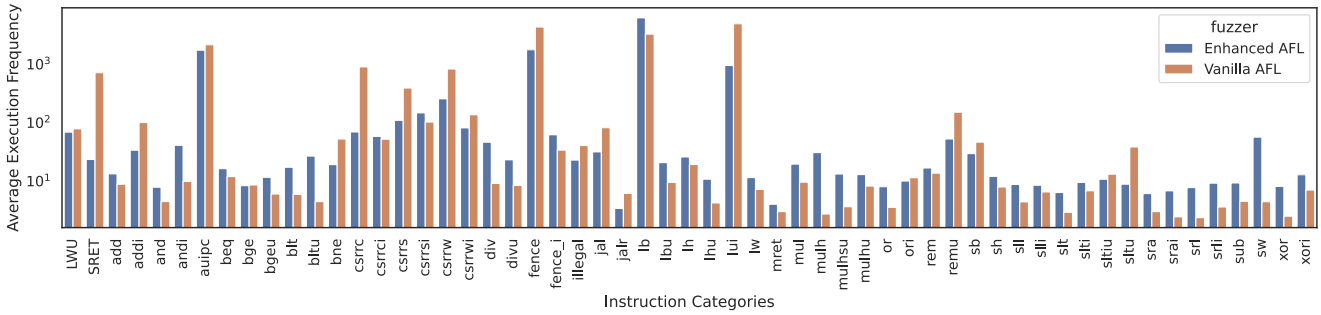

Figure 2: Instruction Execution Frequency

This is illegal because it is not allowed that the complete instruction set is disabled. In summary, we demonstrated that our enhanced mutations improve the fuzzing results statistically significant and we have shown the successful application of our approach by finding 24 bugs in the popular and well-tested RTL-core VexRiscv.

6 DISCUSSION AND FUTURE WORK

Our fuzzing methodology revealed 24 bugs in the well tested RTL-core VexRiscv. Thus, the results demonstrate the applicability of CGF in combination with co-simulation for cross-level processor verification. We improved the verification results of a state-of-the-art fuzzer highly significant by devising a set of optimized fuzzing mutations. For future work we plan to focus on hybrid techniques combining fuzzing with formal verification.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127, within the project VerSys under contract no. 01IW1900, and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

REFERENCES

[1] 2012. RISC-V Torture Test Generator. https://github.com/ucb-bar/riscv-torture.
[2] 2013. RISC-V ISA Tests. https://github.com/riscv/riscv-tests.
[3] 2016. Fuzz testing in Chromium. https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/README.md
[4] 2016. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL
[5] 2018. AFL: Understanding the status screen. https://github.com/google/AFL/blob/fab1ca5ed7e3552833a18fc2116d33a9241699bc/docs/status_screen.txt.
[6] 2018. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[7] 2018. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[8] 2018. Microsoft security development lifecycle. https://www.microsoft.com/en-us/sdl/process/verification.aspx.
[9] 2018. OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz.
[10] 2020. About RISC-V. https://riscv.org/about/
[11] 2020. OneSpin 360 DV RISC-V Verification App. https://www.onespin.com/solutions/risc-v.
[12] 2020. RISC-V Compliance Task Group. https://github.com/riscv/riscv-compliance.
[13] 2020. RISC-V Formal Verification Framework. https://github.com/SymbioticEDA/riscv-formal.
[14] 2020. RISCV-DV. https://github.com/google/riscv-dv.
[15] 2020. VexRiscv. https://github.com/SpinalHDL/VexRiscv Commit: 98de02051e1a5c9400c022dc61acd4bd0507f8a5.
[16] 2021. verilator. https://www.veripool.org/verilator/.
[17] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. 2004. Genesys-Pro: innovations in test program generation for functional processor verification. *IEEE Design & Test of Comp.* (2004), 84–93.
[18] Niklas Bruns, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2021. Toward RISC-V CSR Compliance Testing. *IEEE Embedded Systems Letters* 13, 4 (2021), 202–205.
[19] Niklas Bruns, Vladimir Herdt, Eyck Jentzsch, and Rolf Drechsler. 2022. Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging. In *Design, Automation and Test in Europe*.
[20] Brian Campbell and Ian Stark. 2014. Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation. In *Formal Methods for Industrial Critical Systems*, Frédéric Lang and Francesco Flammini (Eds.). 185–199.
[21] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. 2002. X-Gen: a random test-case generator for systems and SoCs. In *IEEE International High Level Design Validation and Test Workshop*. 145–150.
[22] S. Fine and A. Ziv. 2003. Coverage directed test generation for functional verification using Bayesian networks. In *Design Automation Conf.* 286–291.
[23] Laurent Fournier and Avi Ziv. 2008. Using Virtual Coverage to Hit Hard-To-Reach Events. In *Hardware and Software: Verification and Testing*. Springer Berlin Heidelberg, 104–119.
[24] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Towards Specification and Testing of RISC-V ISA Compliance. In *Design, Automation and Test in Europe*. 995–998.
[25] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V based Virtual Prototype. In *Forum on Specification and Design Languages*.
[26] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing. In *Design, Automation and Test in Europe*.
[27] Vladimir Herdt, Daniel Große, Eyck Jentzsch, and Rolf Drechsler. 2020. Efficient Cross-Level Testing for Processor Verification: A RISC- V Case-Study. In *Forum on Specification and Design Languages*. 1–7.
[28] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. 2011. Feedback-Based Coverage Directed Test Generation: An Industrial Evaluation. In *Hardware and Software: Verification and Testing*, Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz (Eds.).
[29] Y. Katz, M. Rimon, and A. Ziv. 2012. Generating instruction streams using abstract CSP. In *Design, Automation and Test in Europe*. 15–20.
[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.
[31] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *International Conference on Computer-Aided Design*. 1–8.
[32] Weiqin Ma, A. Forin, and Jyh-Charn Liu. 2010. Rapid prototyping and compact testing of CPU emulators. In *RSP*. 1–7.
[33] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *ISSTA*. 261–272.
[34] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* (1990), 32–44.
[35] Katharina Ruep and Daniel Große. 2022. SpinalFuzz: Coverage-Guided Fuzzing for SpinalHDL Designs. In *European Test Symposium*.
[36] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. *CoRR* abs/2102.02308 (2021). arXiv:2102.02308 https://arxiv.org/abs/2102.02308
[37] Harry Wagstaff, Tom Spink, and Björn Franke. 2014. Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators. In *CASES*. 1–10.
[38] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. RISC-V Foundation.
[39] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. RISC-V Foundation.