SPINALFUZZ: Coverage-Guided Fuzzing for SpinalHDL Designs

Katharina Ruep Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
katharina.ruep@jku.at daniel.grosse@jku.at

***Abstract*—Boosting hardware design productivity is a major plus of SpinalHDL, a Scala-based *Hardware Description Language* (HDL). SpinalHDL achieves this by providing object oriented programming, functional programming, and meta-hardware description finally enabling the generation of Verilog code. Despite all the advantages of SpinalHDL, verification is the biggest challenge here as well.**

In this paper, we bring *Coverage-Guided Fuzzing* (CGF), a well-established software testing technique, to the SpinalHDL design flow. We have implemented our approach SPINALFUZZ on top of the fuzzer AFL++. We leverage Scala-features to automate as many tasks as possible and ease the integration of fuzzing in SpinalHDL. In the experiments we demonstrate the effectiveness of SPINALFUZZ in comparison to *Constrained Random Verification* (CRV). For a wide range of SpinalHDL designs we show that SPINALFUZZ outperforms CRV and reaches coverage-closure.

I. INTRODUCTION

Confronted with the end of Moore's law and Dennard scaling, a new golden age for computer architecture has started [1]. However, to make this a reality, domain-specific architectures and designs are essential and, hence, the design productivity needs an enormous boost. Therefore, alternatives to *Register Transfer Level* (RTL) design, which is based on the classical *Hardware Description Languages* (HDLs) Verilog or VHDL, moved in the focus of research. A very promising alternative is SpinalHDL [2]. SpinalHDL has been realized as an embedded *Domain Specific Language* (DSL) in Scala and allows to describe RTL designs by using object-oriented and functional programming. Moreover, the meta-programming features of Scala can be used for parametrization and hardware code generation. In the final step of the SpinalHDL flow, a Verilog design is generated for the design. However, while productivity gains by a factor of 3 or more have been reported using meta-modeling and code generation [3], verification must keep up, since otherwise the verification gap is widening even faster.

In this work, we focus on simulation-based verification of SpinalHDL designs. To define the stimuli for simulation, the verification engineer can either create directed tests or make use of more advanced techniques, like *Constrained Random Verification* (CRV) [4], [5]. The latter has been further improved by integrating coverage feedback from simulations. This, however, requires adjustments of weights and constraints, design-specific Bayesian networks [6], or utilization of data mining techniques [7], each with high manual effort.

In software testing *Fuzzing* is a well-established technique [8]. Fuzzing is a process where the *Program Under Test* (PUT) is executed repeatedly with random-generated inputs to find software bugs and security vulnerabilities. State-of-the-art *Coverage-Guided Fuzzing* (CGF) aims to maximize code coverage of the PUT. The core principle of CGF is as follows: A *corpus* is created which contains at least one initial test case. Then, the CGF *feedback loop* starts, which consists of two main steps: (1) From the corpus a test case is taken, mutated to create a new one and then fed to the PUT. (2) If the overall coverage increases during PUT execution, the new test case is added to the corpus, otherwise it is discarded. With this prioritization of interesting (coverage-increasing) test cases, CGF boosts the efficiency of basic fuzzing significantly.

However, for hardware, only very few fuzzing approaches have been presented so far. None of these approaches target SpinalHDL and they all require a lot of user interaction to setup fuzzing (details see Section II). Moreover, a common challenge for the hardware domain is the additional ingredient of a *harness*. The harness has to (1) translate a test case from the corpus, typically a byte stream, to the input signals of the *Device Under Test* (DUT) and (2) support the sequential behavior of hardware, i.e. input values over time.

To overcome these challenges, we present SPINALFUZZ, a CGF approach for SpinalHDL designs in this paper. During the development of SPINALFUZZ our primary objective was to make fuzzing of a SpinalHDL DUT as easy as possible and, hence, automate as much as possible. For the integration of fuzzing into the SpinalHDL design flow, we (a) leverage the SpinalHDL/Scala language features for various generation tasks and (b) benefit from existing software fuzzers. The main contributions of this paper can be summarized as follows:

- Automatic generation of input corpus and fuzzer harness
- Support of SpinalHDL hardware assertions
- Class-based plugin of SPINALFUZZ for SpinalHDL flow
- Demonstration on a wide range of SpinalHDL designs
- SPINALFUZZ is available as open-source on GitHub[1]

II. RELATED WORK

Motivated by the success of software fuzzing, first hardware-related fuzzing approaches emerged, e.g. fuzzing of firmware [9], [10] and *Instruction Set Simulators* (ISSs) [11].

A first fuzzer targeting HDL designs has been published as RFUZZ [12], where the authors combined fuzzing with FPGA-acceleration. They introduced *Mux Toggle Coverage*, a new RTL coverage metric that measures the values of select signals of multiplexers and use it to provide the fuzzer feedback.

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

The main focus of the work is the implementation on an FPGA and therefore the coverage collection has been realized on the netlist. Recently, DirectFuzz [13] has been presented which extends RFUZZ to generate test inputs maximizing the coverage of a specific block of the hardware design. In contrast, SPINALFUZZ targets the design and simulation flow of SpinalHDL, automates harness and corpus generation and is able to determine coverage without additional logic.

In [14], the authors are focusing on fuzzing of processors. They create a specific grammar and harness to interact with the *TileLink Uncached Lightweight* (TL-UL) interface and integrate them into a hardware fuzzing pipeline. For other designs and interfaces, additional harnesses and grammars need to be created. SPINALFUZZ uses a generic approach and is able to fuzz any SpinalHDL hardware design.

Recently [15] reported that the tool ChiselVerify was extended to enable fuzzing of hardware designs described in Chisel [16]. The presented steps are interesting, however besides manual corpus and harness creation it targets Chisel.

III. PRELIMINARIES

A. CGF & AFL++

Fuzzing was introduced in 1990 by Miller et al. [17] to analyze the reliability of UNIX tools. In the following years an independent research field emerged [8]. Modern fuzzers, like AFL [18] and AFL++ [19] are generally categorized as CGF and work as described in the introduction. AFL++ originated from AFL, targets C++ code, and incorporates state-of-the-art fuzzing research [20]. The coverage feedback metric used in this paper is edge coverage in the form of a compact bitmap, where each edge, i.e. each transition between two basic blocks, is represented by a byte. This metric is highly efficient because the bitmap can be analyzed in microseconds. In addition, the source code of the PUT is analyzed to create a dictionary. This dictionary stores interesting values, such as constants used in if-conditions, and they are used when mutating a test case.

AFL++ implements two types of mutators: deterministic and havoc. Deterministic strategies are single mutations, including bit flips, addition or subtraction of small integers and insertion of interesting integers or values of the dictionary. Havoc strategies define several randomly stacked mutations which change the length of the test case (e.g. by trimming or expanding) or merge two test cases. If a mutated test case leads to new edges in the bitmap representing the PUT, the test case is stored in a queue directory, which acts as the input corpus for further mutations. However, if a test case leads to a *crash* (i.e. unexpected termination) of the PUT, it is stored in a separate crash directory.

B. SpinalHDL

SpinalHDL [2] is a modern Scala-based DSL, which enables to use software paradigms well-established in higher programming languages. Furthermore, SpinalHDL provides the package `spinal.lib` which includes several designs, like bus interfaces, peripherals and timer. Further details and examples can be found in the documentation [2].

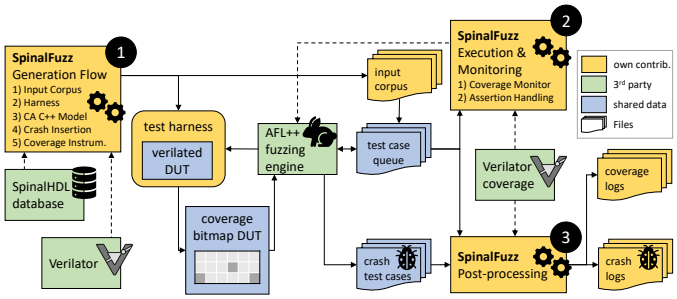

Fig. 1: Overall flow of SPINALFUZZ

As a running example we use a *Greatest Common Divisor* (GCD) design to demonstrate the steps of SPINALFUZZ in Section IV. This design implements the GCD of two positive integers using Euclid's algorithm. The number of cycles needed depends on the two inputs. More cycles are needed if the numbers differ greatly (e.g.: gcd(2,1337)) because the calculation loop needs to be executed more often.

IV. SPINALFUZZ

In this section, we present SPINALFUZZ, our proposed fuzzing approach for SpinalHDL. The overall flow of SPINALFUZZ is depicted in Fig. 1. SPINALFUZZ consists of three key steps. They are shown in Fig. 1 as the yellow rectangles ❶ *Generation Flow*, ❷ *Execution & Monitoring* and ❸ *Post-processing*. To illustrate how SPINALFUZZ interacts with Verilator [21] and AFL++, they are depicted in green, while the shared data (structures) are shown in blue.

Putting it all together, the functionality of SPINALFUZZ is provided to the user as a class-based Scala plugin in the SpinalHDL flow. This enables the verification engineer to easily replace the classical SpinalHDL testbench of the DUT with CGF such that coverage-closure can be reached very fast.

In the following subsections, each key step (including possible subtasks) is described in detail.

A. Generation Flow

In the generation flow step of SPINALFUZZ, depicted as ❶ in Fig. 1, the following five generation tasks are performed:

1) Input Corpus: During fuzzing the test cases (byte streams) from the input corpus are constantly mutated and evolved in search of new interesting, i.e. coverage-increasing, input. SPINALFUZZ generates the initial input corpus based on the input information of the SpinalHDL DUT. The respective initial test case consists of at least one input structure that is built up by the summarized bit widths of the input signals of the DUT. Please note, SPINALFUZZ only ensures that "enough bytes" are available in the initial test case for a single cycle, i.e. the mapping from bytes to concrete DUT inputs is not part of the input corpus. Instead this mapping is done by the harness. In the main fuzzing loop of AFL++, a test case (byte stream) can be enlarged (via mutators) which means that input assignments for several clock cycles will be created and fed to the DUT through the harness. Since a central goal in fuzzing is a fast fuzzing loop, and hence compact/short test cases, we had the idea to add one additional byte after the

Input file rep.	1 byte	2 bytes	3 bytes	4 bytes
Verilator format	CData	SData	IData	QData
C++ equivalent	uint8	uint16	uint32	uint64
Input bit width	1 – 8	9 – 16	17 – 32	33 – 64

Fig. 2: Mapping of input signals in the harness

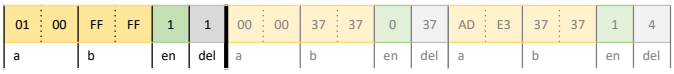

Fig. 3: Example of initial test case (up to the thick line) and expanded test case for GCD with data-width 16

input structure to model a *delay* between consecutive input assignments. As we will show in the experiments this clearly improves the coverage for designs with timers, pipelines or bus interfaces. After discussing the harness, we will provide a concrete example.

2) Harness: SPINALFUZZ generates the test harness that will act as an interface between AFL++ and the later verilated DUT (see Fig. 1 and following Subtask 3). To connect the DUT with the fuzzer, the major tasks of the harness are parsing a test case (i.e. a byte stream which has been selected from the input corpus and mutated) and assigning it to the DUT. To make the harness generation fully automated, SPINALFUZZ uses the same DUT input signal information already determined for the input corpus and adds it to an array in the harness. This array consists of pairs that store input signal references to the later verilated DUT inputs and the number of bytes used to represent each signal. Fig. 2 shows the mapping from the byte stream to the DUT input signals inside the harness. The first row describes the number of bytes that are stored in the array and also acts as link to the test cases of the input corpus. The second and third row show the corresponding Verilator format and its C++ equivalent, respectively. The last row provides the link to the DUT input signals and shows the range of bits that are covered by the corresponding representations.

Example 1. *We consider the* GCD *design: The input corpus generated by* SPINALFUZZ *consists of two bytes for each data signal* a *and* b*, and one byte for the enable signal* en*. Additionally, a delay is used since the implementation requires several clock cycles for the calculation.*

In Fig. 3 we provide a concrete example. The first 6 *bytes (up to the thick line) are the input corpus that has been generated by* SPINALFUZZ *(input data width for* a *and* b *is* 16*, values in hex). The following* 12 *bytes in Fig. 3 (shown lighter) are determined by AFL++ during fuzzing in key step* ❷*. In other words, the initial test case is evolved by advanced AFL++ mutations to increase the overall coverage.*

3) Cycle-Accurate C++ Model: After the definition of the interface between input corpus and harness as well as the generation of the harness itself, the DUT is now "verilated", i.e. it is transformed into a cycle-accurate C++-model. For this we use Verilator [21] and the harness becomes the entry point for the DUT.

4) Crash Insertion: If there are user-defined hardware assertions inside the SpinalHDL DUT, SPINALFUZZ has to adapt the runtime behavior of the verilated DUT: Instead of just aborting the simulation, we take advantage of the crash-detection ability of the fuzzer and emit a program crash.

5) Coverage Instrumentation: AFL++ is providing a wrapper for compilation to control the coverage instrumentation. As we are only interested in the coverage-progress of the DUT, coverage instrumentation is only performed on the C++ files representing the DUT. Thus, harness and Verilator's runtime library are not taken into account for the fuzzer's internal coverage feedback.

B. Execution & Monitoring

The Execution & Monitoring step, shown as ❷ in Fig. 1, focuses on the execution of AFL++ which requires parallel coverage monitoring and appropriate handling of assertions.

Analogous to the general fuzzing loop of CGF (as described in Section I and Section III-A), the fuzzing loop of SPINALFUZZ works as follows: execution of the DUT with an initial (or later mutated) test case, collection of coverage from the compiled DUT based on the performed AFL++ coverage instrumentation, storage of coverage-increasing and crash-generating test cases, and mutation of coverage-increasing test cases that are passed to the executable again. In Fig. 1 the main fuzzing loop starts at AFL++, continues to the harness and the coverage bitmap of the DUT, and ends in AFL++ again.

1) Coverage Monitoring: In addition to the AFL++ fuzzer execution loop, SPINALFUZZ also supports periodic tracking of Verilator coverage. This is done by monitoring the queue directory, where AFL++ is storing coverage-increasing test cases, and extracting the Verilator coverage of new test cases. The reason for this choice is that Verilator coverage is hardware-oriented, can be back-annotated to Verilog (which is beneficial for traceability and debugging; see also Section IV-C) and allows a fair comparison to CRV. We use the strongest coverage metric from Verilator, i.e. basic block line coverage (utilizing unique counters at each code flow change point, which are the branches of IF and CASE statements).

2) Assertion Handling: Based on SPINALFUZZ's crash insertion described in Section IV-A4 and AFL++'s crash monitoring, AFL++ is able to find violations of user-defined SpinalHDL assertions. If AFL++ is reaching such a violated assertion, the execution will crash and then be recorded. Note that AFL++ will only record unique crashes and will store the triggering test cases in the crashes directory. AFL++ defines crashes as unique if new state transitions are reached with the current execution, which no other recorded crash has reached before. The unique crash feature of AFL++ is a clear advantage over CRV because the amount of failing test cases is greatly reduced to only representative test cases originating from different execution behavior.

Finally, CGF ends if the chosen end condition is satisfied: *timeout*, *full coverage* or *assertion violation*.

C. Post-processing

To get the most out of fuzzing, two reports are generated (Step ❸ in Fig. 1). First, Verilator coverage is extracted for

TABLE I: Evaluation results for SPINALFUZZ (SF) and CRV

Benchmark	Design spec.			Cov. [%]		Sat. [s]	
	In	Cells	FFs	CRV	**SF**	CRV	**SF**
GCD	33	392	34	**97**	**97**	50	2
CNN-Buffer	4	164	24	63	**96**	1	5
Alu	69	685	0	80	**100**	1	5
I2cSlave	41	275	53	77	**95**	161	20
Apb3Timer	45	340	62	75	**94**	1	26
SpiXdrMaster	77	491	38	89	**99**	1	27
Apb3SpiSlave	46	1,509	583	66	**92**	1	5
UartCtrl	102	476	69	77	**96**	1,623	30
Apb3UartCtrl	41	1,344	394	62	**94**	3,461	673
BmbI2cCtrl	56	682	149	79	**93**	1,655	3,556

all coverage-increasing test cases of AFL++. Second, for the crash report Verilator coverage is extracted for all crash-generated test cases. This improves debugging because the input resulting in a crash and therefore violating an assertion is provided for each *unique path* to the violated assertion. Furthermore waveforms, annotations and procedure logs are generated for each test case. This provides more details from a hardware oriented perspective.

V. EXPERIMENTS

All experiments have been performed on an Ubuntu 21.04 system with an Intel i7-10510U processor and 32 GB memory.

We consider several benchmarks from (a) the official `spinal.lib` package as well as (b) custom designs. For all these benchmarks we focus on the effectiveness of SPINALFUZZ, i.e. we compare the progress of coverage of SPINALFUZZ vs. CRV. All designs were run with 1 hour timeout. The results are summarized in Table I. Benchmarks from *Alu* to *BmbI2cCtrl* are designs from the `spinal.lib` package. They include arithmetic designs, communication blocks, timer and peripherals. The benchmark *GCD* is the custom design introduced in Section III-B with a *dataWidth* of 16 bit and *CNN-Buffer* [22] belongs to an AI accelerator. The first column lists the name of the benchmark. The next three columns provide some insight in the complexity of the benchmarks. Here, input bits (In) are summed up and are followed by the overall cells and *Flip-Flops* (FFs) generated with the *synth*-command of *yosys* [23]. The last four columns show the Verilator coverage after one hour in percent and the time needed until saturation in seconds for both, SPINALFUZZ and CRV. As can be seen, SPINALFUZZ always reaches more than 90% coverage. Except one benchmark (*GCD*) the coverage achieved by SPINALFUZZ is much higher. For this exception SPINALFUZZ reaches the same coverage much faster (2 seconds vs. 50 seconds). Looking at the saturation time, we can observe that CRV sometimes reaches good coverage values (approx 70%) also very fast, but then gets stuck. Here only manual interaction by adjusting weights or adding additional constraints would improve the coverage. For instance for the benchmark *Alu*, a purely combinational design, SPINALFUZZ even reaches 100% coverage in 5 seconds, whereas *CRV* can't increase its 80% coverage within the time-out limit. The smallest and largest designs, according to *design specifications* columns, are *CNN-Buffer* and *Apb3SpiSlaveCtrl*. For both designs the greatest improvement in coverage in comparison to *CRV* is reached with almost 30% difference. In column *Sat.* (Saturation) it can be seen that SPINALFUZZ only needs seconds to reach its highest coverage with the exception of benchmark *BmbI2cCtrl*. Even in this case SPINALFUZZ needed only 364 seconds (not shown in the table) to reach *CRV*s final coverage of 79%. For all benchmarks applies that if SPINALFUZZ reaches its final coverage later than *CRV*, the resulting coverage is much higher.

VI. CONCLUSIONS

In this paper we proposed SPINALFUZZ, a fully automated CGF approach for SpinalHDL designs. In the experiments we demonstrated the effectiveness of SPINALFUZZ in comparison to CRV on a wide range of SpinalHDL designs. In all cases SPINALFUZZ outperformed CRV and reached a coverage greater than 90%.

For future work we plan to investigate the implications of unique crashes for debugging as well as the integration of our approach with programmable waveform analysis [24].

ACKNOWLEDGMENTS

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

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Comm. of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.
[2] "SpinalHDL," https://github.com/SpinalHDL.
[3] W. Ecker and J. Schreiner, "Metamodeling and code generation," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Springer, 2017, pp. 1–41.
[4] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.
[5] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *SoC*, 2012, pp. 1–7.
[6] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003, pp. 286–291.
[7] C. Ioannides and K. I. Eder, "Coverage-directed test generation automated by machine learning – a review," *ACM Trans. on Design Automation of Electronic Systems*, vol. 17, no. 1, pp. 7:1–7:21, 2012.
[8] V. J. Manès *et al.*, "The art, science, and engineering of fuzzing: A survey," *TSE*, vol. 47, no. 11, pp. 2312–2331, 2021.
[9] M. Muench *et al.*, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS*, 2018.
[10] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *USENIX Security Symposium*, 2019.
[11] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.
[12] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *ICCAD*, 2018, pp. 1–8.
[13] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *DAC*, 2021, pp. 529–534.
[14] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," 2021, arXiv:2102.02308.
[15] A. Dobis, T. Petersen, and M. Schoeberl, "Towards functional coverage-driven fuzzing for chisel designs," in *WOSET*, 2021.
[16] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012.
[17] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Comm. of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[18] "AFL - American Fuzzy Lop," https://github.com/google/AFL.
[19] "AFL++ - American Fuzzy Lop++," https://github.com/AFLplusplus/AFLplusplus.
[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *WOOT*, 2020.
[21] "Verilator," https://www.veripool.org/verilator.
[22] L. Klemmer, S. Froehlich, R. Drechsler, and D. Große, "XbNN: Enabling CNNs on edge devices by approximate on-chip dot product encoding," in *ISCAS*, 2021, pp. 1–5.
[23] "yosys - Yosys Open SYnthesis Suite," https://github.com/YosysHQ/yosys.
[24] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.