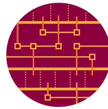
P5: Automated Generation of System-Level Test Programs for Characterization of Parametric Device Properties

Milestone Report – GS-IMTR (for internal use only)

Denis Schwachhofer

December 1, 2022



Advisor / main examiner: Prof. Dr. rer. nat. habil. Ilia Polian

Co-examiner: Prof. Dr. rer. nat. Dirk Pflüger

Mentor: Dr. Matthias Sauer

1 Introduction

System-Level Test (SLT) is increasingly used to improve quality assurance of complex Systems-on-Chip (SoCs). In contrast to earlier test insertions based on structurally generated test patterns, SLT aims at detecting defective chips by approximating the end-user environment of an SoC by, for example, putting a smartphone SoC into a test board that closely resembles a smartphone. Today, SLT uses off-the-shelf software such as operating systems or applications typically executed in the Device Under Test's (DUT's) mission mode [1]. To expand on the example above, one would boot up Android™ and run some apps, such as a browser or a video player. Test engineers manually select these workloads. During SLT, the DUT is usually considered a blackbox with an unknown gate-level structure; occasionally, a party with no knowledge of the DUT's netlist, e.g., the integrator, even applies SLT. This restriction applies due to some IP (Intellectual Property) cores being used in an SoC that are only available as blackboxes with just their I/O interfaces and behavioral descriptions.

There are several problems associated with this process: Typical SLT runtimes are around 15 minutes to 2 hours, whereas traditional testing for even complex SoCs takes at most a minute. Furthermore, as mentioned, the test suite is hand-picked by the test engineers, and possibly even the execution of this test is done manually. The missing automation increases test time and cost even more. And finally, besides simple pass-fail, there is no metric to determine if the selected programs form a high-quality test suite, i.e., can detect as many defects as possible. The only guidance test engineers have when composing such test suites are fallout rates during SLT and the analysis of test escapes.

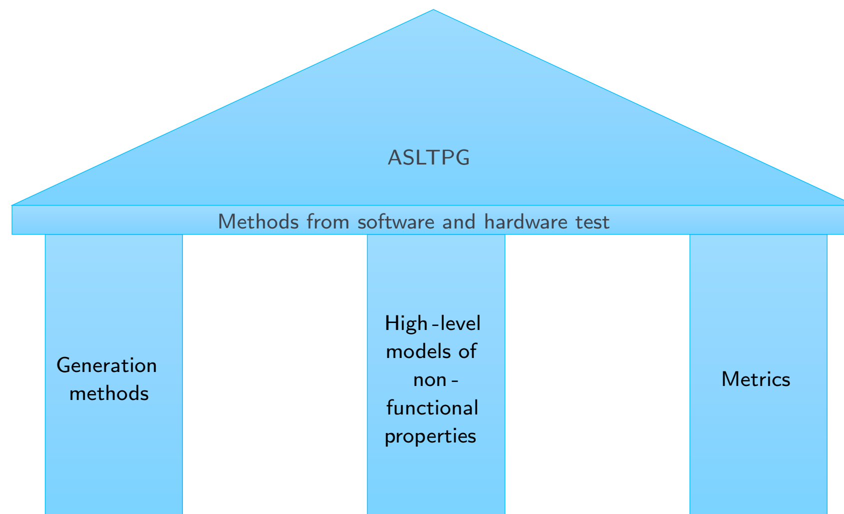


Figure 1: The three pillars of ASLTPG

Project P5 aims to develop a tool that can tackle the aforementioned problems of SLT called ASLTPG (Automatic System-Level Test Program Generator). The project itself is based on the following three pillars (as seen in Figure 1):

- Exploring generation methods
- Creating a high-level model of non-functional properties
- Finding new metrics

Additionally, we want to consider not only methods and concepts from hardware test but also from software testing as well.

Generation Methods There are existing methods to generate assembly code snippets, for example, MicroGP [2]. We want to evaluate these techniques and also find out if techniques from software testing, for example, fuzzing, might be feasible and compare them against each other. The generation method we will choose must be highly scalable and also be able to optimize a test program under multiple objectives: non-functional and functional properties and possibly the coverage of SLT unique fails.

Since our DUT is seen as a blackbox, we also need the generation method to be able to provide high-quality

results without relying on details about the architecture of the DUT. As such, traditional methods based on Automatic Test Pattern Generation (ATPG) cannot be used, since they rely on structural information. From our perspective, the optimal method would generate snippets or a test suite in a higher-level language, for example, C. It would also only incorporate necessary knowledge, such as the memory map and which periphery is available where, and how to interact with it. This information can, for example, be provided via the Portable Test and Stimulus Standard [3].

High-level Model of Non-functional Properties A high-level model of non-functional properties allows us to improve test generation speeds since we then are able to bypass long and expensive simulations or even actual hardware measurements. Such a model can be created using Machine Learning, for example. Such non-functional properties entail the power consumption of the DUT and its temperature, but also delay models. We can combine these hard-to-obtain non-functional properties with others that can be obtained via RTL simulation. Performance counters or software telemetry, for example, the performance of an algorithm or a benchmark score, can be used for this purpose.

Metrics Finally, due to the blackbox assumption, we cannot use traditional fault models as well. On top of that, fault grading for a complex SoC is already time-consuming for traditional test patterns. And the current methods do not scale well for SLT programs which typically consist of much more patterns. As such, it would be infeasible to measure fault coverage of SLT programs even if we could use traditional fault models.

Thus, we need new metrics to grade the quality of our test programs. These new metrics might either be related to the test program itself. Or they can be obtained by running RTL simulations. One idea might be to monitor bus transactions and either cause maximum bus contention or instead try to cover as many unique transactions as possible. More ideas and research will also be contributed by work in conjunction with project P1, which is looking into the SLT-unique defects and how we can specifically target them.

State of the Art Research on SLT only recently started to gain traction with [1]. The industry adopted SLT a decade ago [4]. [1, 5] and [6] identify the need for SLT and point out the areas where more research is necessary, including the need for methods of test program generation. Tipparthi and Kumar [7] introduced a method to identify mutually exclusive test cases and run them in parallel. Almeida et al. [8] demonstrated an application of SLT where they combined it with Burn-In. Liu and Ou [9] examined how to implement adaptive SLT where only a select subset of ICs is undergoing SLT in order to avoid the long run time and high costs it incurs.

2 Fuzzing for System-Level Test Generation

The following work is a collaboration with project P8 and about to be submitted to a conference. SLT programs are expected to exercise execution paths and transactions that are unlikely to occur during structural test and therefore detect defects not covered by traditional test patterns [5]. However, some defects manifest themselves only under specific extra-functional conditions, e.g., in certain temperature ranges or when the DUT is subject to electrical noise. Therefore, SLT workloads that actively control extra-functional parameters of the DUT are of interest. However, composing them by hand is challenging, particularly in the absence of structural information.

To tackle this problem, we propose a method based on mutation-based greybox fuzzing that is able to generate assembly snippets that control extra-functional properties, such as the power consumption of our DUT. In the following sections, we will introduce mutation-based greybox fuzzing followed by our suggested method. Afterwards, we present the experiment we conducted to evaluate our method and present some results.

2.1 Mutation-based Greybox Fuzzing

Fuzzing is a technique in software testing to generate random or semi-random inputs to detect bugs or possible attacks [10]. It can also be adapted for other goals, e.g., finding inputs with the worst-case run time [11].

The basic idea behind fuzzing is to apply as many different and unique inputs as possible and observe the behavior of the Software Under Test (SUT). Mutation-based greybox fuzzing uses a feedback metric to guide the input generation. Typical metrics are line or branch coverage [12]. AFL++ [13], for example, uses edge coverage, i.e., how often an input causes a transition between basic blocks.

Figure 2 shows a flow chart for a typical greybox fuzzing run. The fuzzer first selects a random input, called seed, from the initial seed corpus. It then applies random mutations to it, for example, flipping bits, trimming, or combining two seeds. Then, the SUT is run with the new input, and the feedback value is collected during execution. The new input will be added to the seed corpus if:

- either it detects new paths and transitions
- or it passes transitions that are already encountered more often

This loop runs until a time limit is reached or the user aborts the fuzzing campaign.

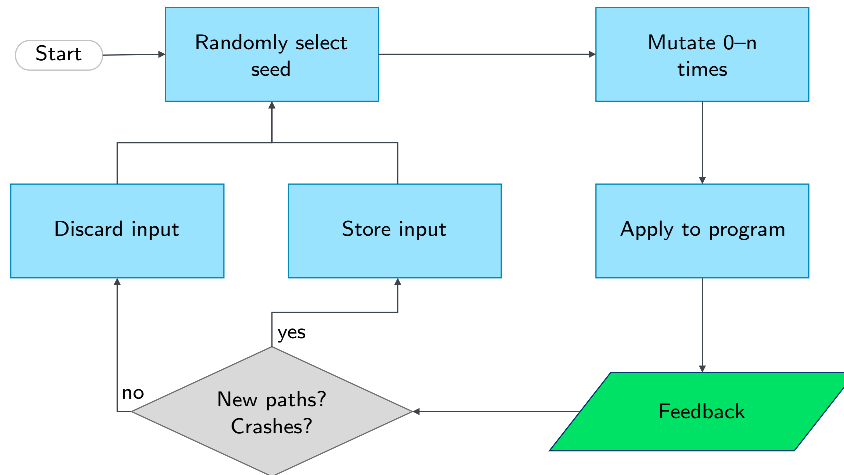


Figure 2: Execution loop in mutation-based greybox fuzzing

We chose fuzzing as a middle ground between exploration, e.g., exhaustive random search, and exploitation, e.g., genetic algorithms or manual test composition. Fuzzing explores its search space freely but uses the feedback value to converge gradually toward more optimal solutions. Exhaustive random search will, at some point, find an optimal solution as well but not in a feasible time.

Also, fuzzing has started to gain traction in hardware design verification. For example, there is *RFuzz* [14]. It uses greybox fuzzing with *mux control coverage* as the feedback. Mux control coverage is defined as the percentage of select signals of multiplexers that have been toggled in a single test. Additionally, they implemented an FPGA acceleration flow to improve fuzzing speed. *DirectFuzz* [15] builds on top of RFuzz but is intended to generate tests for a specific block instead of the whole design. Besides these language-agnostic fuzzers, some others use a specific hardware description language, for example, *SpinalFuzz* [16] which is based on *SpinalHDL* and *ChiselVerify* [17], which is based on *Chisel*.

These fuzzers have in common that they all use functional metrics to guide the fuzzer, and their goal is to detect bugs in hardware designs. Instead, our work uses a non-functional metric, namely power consumption, to generate test programs for SLT. Additionally, we are working with code snippets rather than input vectors and testbenches.

To our knowledge, we are the first to work on methods for SLT program generation and the first to use fuzzing in this application.

2.2 Optimizing Non-functional Properties Using Fuzzing

To adopt greybox fuzzing for our purposes, we developed a new method that uses a proxy to facilitate communication between a fuzzer and the DUT or method for obtaining non-functional metrics. Figure 3 shows the general flow of our new method. The left side shows the fuzzing flow from above, and the right side shows a high-level overview of the steps our proxy executes to provide feedback to the fuzzer. We assume that the fuzzer provides us with code snippets that we have to compile to execute them on

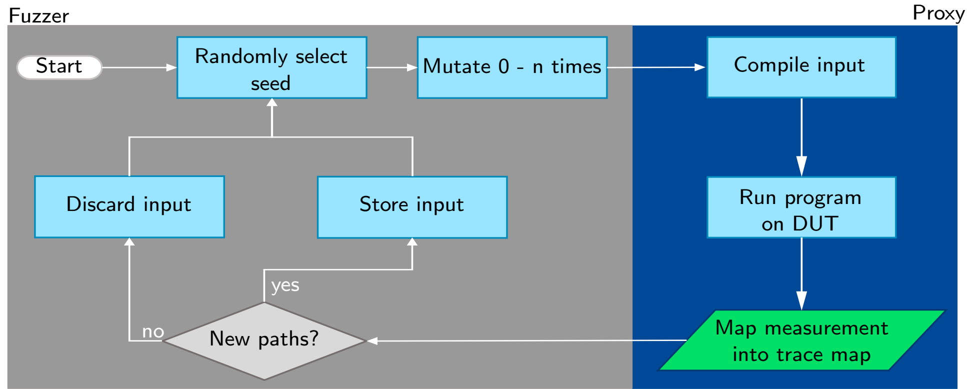


Figure 3: Execution flow of our fuzzer

our DUT. We then apply the compiled snippet to our DUT and collect the measurement. The fuzzer is generally unable to use this measurement as feedback directly, so we have to map it to a format that the fuzzer understands.

The proxy connects the fuzzer with our DUT and measurement methods and thus has to add as little overhead as possible to keep the overall execution time as low as possible. Regarding the mapping of the measurements, one has to bear in mind that, generally, fuzzers are trying to achieve the maximum possible coverage. They can employ different strategies, called power schedules, to decide which seed to select next. Böhme et al. [18] present some of these strategies in their paper. But many strategies boil down to selecting seeds that just detected but didn't exercise new transitions or coverage points. As such, the mapping has to be designed in a way to exploit the power schedule.

2.3 Maximizing Power Consumption with AFL++ and Hardware Measurements

In this section, we will present the experiment to evaluate our method. The goal of this experiment was to generate assembly snippets that consume as much power as possible. Our DUT is a 64-bit, super-scalar RISC-V processor called BOOM [19] with three execution units and no L2 cache running at 25 MHz. It is running on a Xilinx VC707 evaluation board. We connected an SR560 low-noise amplifier to a 0.5 mΩ shunt resistor connected to the FPGA's voltage rail and to a Rohde & Schwarz RTM3004 oscilloscope to collect current measurements. The oscilloscope is connected to a PC via USB and communicates using UART. We use AFL++ as our fuzzer.

2.3.1 The Experiment

AFL++ applies mutations to bitstreams, so we have to introduce another tool to convert those to actual valid assembly snippets. For this purpose, we use a tool developed by P8. It uses a grammar to decode these bitstreams to mostly syntactically valid RISC-V assembly snippets. Some snippets might be invalid because the grammar does not allow us to control the constant values that are decoded, thus causing compilation to fail. But, as we will show later, the number of invalid snippets is small compared to the total amount.

We then compile these snippets and load them into the memory of our DUT via GDB and OpenOCD. Our first measurement is taken while our DUT is in a halt state. Afterwards, we tell the DUT to execute the snippet in an infinite loop and take the second measurement. The measurement that will be mapped to AFL++'s trace map is the difference between run and idle measurement. The reason we need to take two measurements is that we want to reduce the influence of ambient and core temperature on the measurements.

AFL++'s trace map is a byte array where each index represents a transition between two locations. In our experiment, these locations are defined as classes of instructions. The following classes exist:

- ALU (`mv`, ...)
- MEM (`ld`, ...)

Table 1: Mapping of measurements to bins

Bin	% of max_{cur}
0	$< 50\%$
2	50–60%
4	60–80%
6	80–100%
8	$> 100\%$

- ALUIMM (`li`, ...)
- FP (`fadd.d`, ...)
- FMEM (`flw`, ...)
- MULDIV (`mul`, ...)

The generated snippets will result in different byte arrays depending on the class and the order in which they occur. For example, a `mv` instruction followed by an `ld` instruction will set the byte at the index that corresponds to the transition ALU $\rightarrow$ MEM.

We set the byte at the index according to the measurement itself. For this, we calculate the relative measurement compared to the current maximum measurement max_{cur} of the fuzzing campaign. Table 1 shows the mapping of the measurements into the respective bins. Bin 0 means that we throw away the input. Otherwise, the most significant bit will be set according to the bin of the measurement. With this mapping, we accomplish the following: 1) Explore a diverse set of inputs by using transitions between classes of instructions, and 2) optimize input generation towards higher power consumption by setting the most significant bit accordingly.

Additionally, there is a repeatability requirement to avoid faulty measurements negatively impacting the experiment: A snippet is only accepted as a new optimum when it supersedes the current one r_{bin8} times in a row.

2.3.2 Results

The fuzzing campaign and the previously mentioned experiment were executed on a machine running CentOS 7. The machine has an Intel® Core™2 Quad running at 2.83 GHz and 8 GB of RAM. We used version 4.01a of AFL++. We set r_{bin8} to 2. The initial seed corpus consists of two hand-crafted snippets, which we later use as a comparison as well.

We let the fuzzing campaign run for 74 hours. During the campaign, AFL++ generated 28,388 unique snippets and executed 70,659 runs. It kept 4348 in the seed corpus. 1369 snippets failed compilation. AFL++ reported stability to be around 70%.

AFL++ reapplies some inputs it determined as interesting multiple times back-to-back. This is called calibration and helps AFL++ determine average execution speed and detect anomalies as well. During this step, it also measures the stability of the input, i.e., does the same input always result in the same feedback? High stability is required for the fuzzer to understand if the inputs have an actual impact or if the feedback is just noise. Future work might use the results from this campaign to determine fixed thresholds to increase stability.

For this experiment, it is impossible to reach 100% stability due to measurement tolerances. Additionally, if a new optimal snippet has been found, all the thresholds for the bins shift upwards as well, causing inconsistent feedback values.

The average execution speed was 0.23 exec/s, i.e., a single execution took around 4.3 s. The limiting factor for the execution speed is the hardware setup: Loading the binary onto the FPGA, instructing the processor to run the snippet and getting the measurements from the oscilloscope take up most of the time a single execution needs.

Figure 4 shows all measurements taken during the fuzzing campaign. It shows that fuzzing is more explorative than exploitative. This is indicated by the measurements being spread out over all bins during most of the campaign. An exploitative method would show a clear trend in the measurements with only

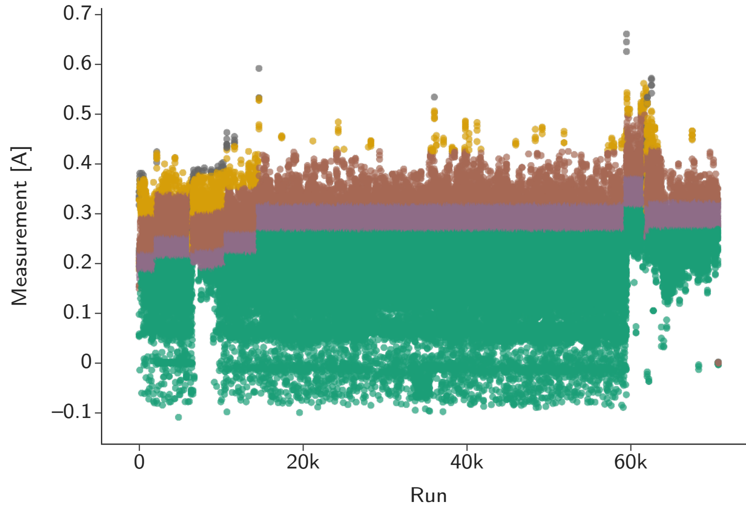


Figure 4: Measurements of fuzzing campaign. Dark green corresponds to bin 0, dark violet to bin 2, brown to bin 4, orange to bin 6, and dark grey to bin 8.

minimal variation. Nonetheless, the influence of the feedback value on the fuzzing campaign is also visible via the general upwards trend in the measurements.

Our fuzzer found many best-performing snippets in the first 15,000 executions. This is indicated by the thresholds for each bin shifting upward. Because of the repeatability condition introduced above, our fuzzer only considers the r_{bin8}-th measurement, which can be lower than any other beforehand. After almost 60,000 executions, our fuzzer finds the best-performing snippet of the campaign.

It should also be noted that every time the thresholds move down in the figure, the fuzzer has been restarted due to an issue with the JTAG interface. AFL++ can recover its state from a restart by re-running all snippets that were in the seed corpus at that moment.

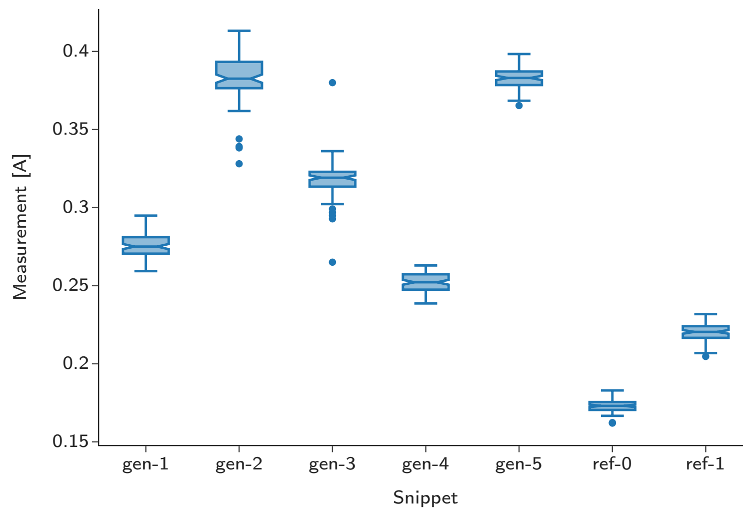


Figure 5: Results for evaluation of snippets. *gen-x* are the generated snippets from the fuzzing campaign, and *ref-x* are the handcrafted snippets.

To evaluate the results we chose the five best snippets and the two hand-crafted snippets and took 100 measurements in a row. The results can be seen in Figure 5. All the generated snippets clearly outperform

the hand-crafted ones. There is also a distinct difference between the generated snippets, with *gen-4* being the worst and *gen-2* and *gen-5* performing the best.

```
1 mv x9, x1
2 mv x2, x9
3 fcvt.lu.s x2, f22
4 fcvt.lu.s x2, f23
5 mv x5 ,x2
6 mv x3 ,x5
7 fmadd.s f24,f22,f27,f21
8 mv x0, x3
```

Listing 1: One of the generated snippets (gen-5)

Listing 1 shows the best-performing snippet *gen-5*. All other generated snippets are quite similar to *gen-5*. They differ mainly in the order of instructions and in the selected registers. *gen-4* is also the only snippet that contains a memory instruction.

To make more general statements about the generated snippets we will look at their composition and length. This will possibly give us insights into why these snippets perform the way they do.

We first look into their composition. Figure 6 shows the distribution for each class of instruction. Since bin 8 only contains 12 snippets, the significance of the values for bin 8 is relatively low, which must be kept in mind.

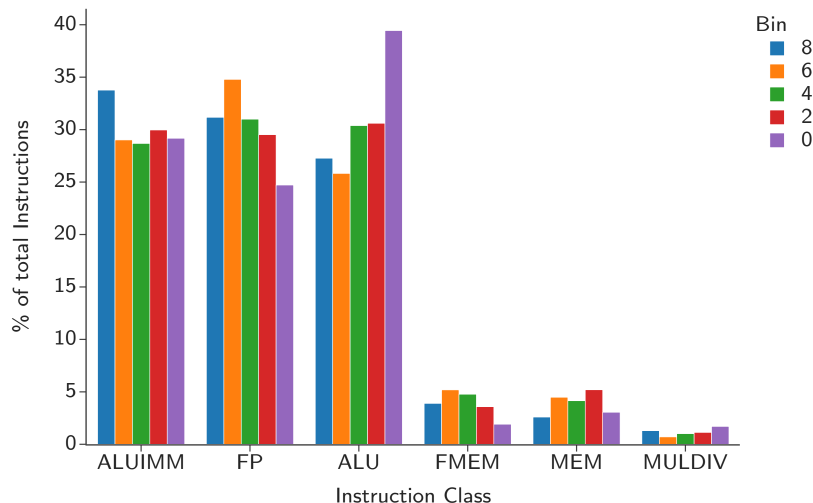


Figure 6: Distribution of instruction classes for each bin

ALUIMM is the dominant class for bin 8 with a percentage of 33.77%, closely followed by the FP class with 31.17% and the ALU class with 27.27%. For bin 0, the distribution is different: ALU is the dominant class with 39.44%, followed by ALUIMM with 29.17% and FP with 24.72%. In general, one can see that including more floating point instructions does increase the power consumption of a snippet. Furthermore, it seems that more ALU instructions instead decrease power consumption. The ALUIMM class does not seem to influence the performance significantly, except for bin 8. Bins 2, 4, and 6 contain more MEM and FMEM instructions than bins 8 and 0. When ignoring bin 8, one can observe that more MEM and FMEM instructions and fewer MULDIV instructions tend to perform better.

Overall, a good snippet consists of a mixture of mainly ALUIMM and FP class instructions. In contrast, the MULDIV class instructions do not positively contribute to power consumption. still, these statements may only hold true for this particular fuzzing campaign.

Now we will look if the length of the snippet plays an important role.

Table 2: Statistics about the number of instructions per bin

Bin	Median	Average	Min	Max
0	8	6.49	1	309
2	9	9.9	1	95
4	8	9.31	1	158
6	8	11.2	1	293
8	6	6.42	3	10

Table 2 shows the median and average amount of instructions per snippet for each bin. It also shows the shortest and the longest snippet per bin. These values have been calculated over all snippets generated in the fuzzing campaign. The median does not differ significantly between the bins, with bin 8 and bin 2 being the furthest apart by three instructions. The average length does vary more, which is mainly due to outliers. For example, bin 6 contains snippets with 298 and 204 instructions. Bin 0 contains snippets with 309 and 201 instructions and many snippets with only a single instruction. For bin 8 the amount of instructions is more consistent.

We also calculated the Pearson correlation coefficient between the number of instructions per snippet and the bin the snippet landed in. It shows a slightly moderate, positive correlation with an r-value of 0.315. Most smaller snippets, less than ten instructions, land in bin 0. The shortest snippet in bin 8 has three instructions, whereas the shortest snippets in every other bin only have a single instruction. On the other hand, one can see that snippets with a large number of instructions do not perform better than shorter ones.

In summary, a snippet's length does contribute to its power consumption but only to a limited degree. We assume that there is an ideal range, but, in general, shorter snippets between 3 and 10 instructions tend to perform better.

3 Conclusion and Next Steps

We have shown that fuzzing is indeed feasible for generating SLT programs and controlling non-functional properties, in this case, power consumption.

We have presented the current state of the project regarding the pillar of exploring generation methods. Further plans for this pillar are including comparisons with other methods, such as Genetic Programming or Reinforcement Learning, and determining which one is the most suitable or if there is a need for a new method or a mix of methods. Also, we will include more properties to optimize for and continue reevaluating these methods. Finally, in collaboration with P8, we will be working to improve the decoder to also generate semantically valid snippets. This will allow us more freedom for snippet generation and eventually also inclusion of periphery, such as UART or Bluetooth, as well.

Currently, work is going on to train a Machine Learning model for the prediction of the power consumption of assembly snippets. This work contributes to the pillar of high-level models. The goal is to either completely replace power simulations or at least supplement them. Power simulation of complex designs takes a very long time and is thus unsuitable for generation methods such as fuzzing. This work is done in collaboration with project P10.

Another collaboration with students of the Politecnico di Torino is in the planning stage and will make contributions to the pillar of metrics.

References

[1] Harry H. Chen. Beyond structural test, the rising need for system-level test. In *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4. IEEE, April 2018. doi: 10.1109/VLSI-DAT.2018.8373238. ISSN: 2472-9124.

[2] Ernesto Sánchez, Massimiliano Schillaci, Matteo Sonza Reorda, Giovanni Squillero, Luca Sterpone, and Massimo Violante. New evolutionary techniques for test-program generation for complex

microprocessor cores. In *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05*, GECCO '05, pages 2193–2194, New York, NY, USA, June 2005. Association for Computing Machinery. ISBN 978-1-59593-010-1. doi: 10.1145/1068009.1068370.

[3] Portable test and stimulus standard version 2.0. *Accellera Standard*, 2021. URL `https://accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf`.

[4] Sounil Biswas and Bruce Cory. An Industrial Study of System-Level Test. *IEEE Design & Test of Computers*, 29(1):19–27, February 2012. ISSN 1558-1918. doi: 10.1109/MDT.2011.2178387. Conference Name: IEEE Design & Test of Computers.

[5] Ilia Polian, Jens Anders, Steffen Becker, Paolo Bernardi, Krishnendu Chakrabarty, Nourhan El-Hamawy, Matthias Sauer, Adit Singh, Matteo Sonza Reorda, and Stefan Wagner. Exploring the Mysteries of System-Level Test. In *2020 IEEE 29th Asian Test Symposium (ATS)*, pages 1–6. IEEE, November 2020. doi: 10.1109/ATS49688.2020.9301557. ISSN: 2377-5386.

[6] Paolo Bernardi, Marco Restifo, Matteo Sonza Reorda, Davide Appello, Claudia Bertani, and D. Petrali. Applicative System Level Test introduction to Increase Confidence on Screening Quality. In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–6. IEEE, April 2020. doi: 10.1109/DDECS50862.2020.9095569. ISSN: 2473-2117.

[7] Dilip Kumar Reddy Tipparthi and Karthik Krishna Kumar. Concurrent system level test (CSLT) methodology for complex system-on-chip. In *2014 IEEE 16th Electronics Packaging Technology Conference (EPTC)*, pages 196–199, December 2014. doi: 10.1109/EPTC.2014.7028421.

[8] F. Almeida, Paolo Bernardi, D. Calabrese, Marco Restifo, Matteo Sonza Reorda, Davide Appello, Giorgio Pollaccia, Vincenzo Tancorre, Roberto Ugioli, and Gulio Zoppi. Effective Screening of Automotive SoCs by Combining Burn-In and System Level Test. In *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–6. IEEE, April 2019. doi: 10.1109/DDECS.2019.8724644. ISSN: 2473-2117.

[9] Chenwei Liu and Jie Ou. Smart Sampling for Efficient System Level Test: A Robust Machine Learning Approach. In *2021 IEEE International Test Conference (ITC)*, pages 53–62, October 2021. doi: 10.1109/ITC50571.2021.00013. ISSN: 2378-2250.

[10] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. URL `https://www.fuzzingbook.org/`.

[11] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, July 2018. Association for Computing Machinery. ISBN 978-1-4503-5699-2. doi: 10.1145/3213846.3213874.

[12] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, November 2021. ISSN 1939-3520. doi: 10.1109/TSE.2019.2946563. Conference Name: IEEE Transactions on Software Engineering.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[14] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018. doi: 10.1145/3240765.3240842.

[15] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021. doi: 10.1109/dac18074.2021.9586289.

[16] Katharina Ruep and Daniel Grose. SpinalFuzz: Coverage-guided fuzzing for SpinalHDL designs. In *2022 IEEE European Test Symposium (ETS)*, pages 1–4. IEEE, 2022. ISBN 978-1-6654-6706-3. doi: 10.1109/ets54262.2022.9810421.

[17] Dobis, Andrew, Petersen, Tjark, and Schoeberl, Martin. Towards functional coverage-driven fuzzing

for chisel designs. In *Workshop on Open-Source EDA Technology (WOSET 2021)*. ETH Zurich, 2021. doi: 10.3929/ETHZ-B-000539444.

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016. doi: 10.1145/2976749.2978428.

[19] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *Fourth Workshop on Computer Architecture Research with RISC-V*, page 7, May 2020.