An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study

Author(s):
Schiavone, Pasquale D.; Sanchez, Ernesto; Ruospo, Annachiara; Minervini, Francesco; Zaruba, Florian; Haugou, Germain; Benini, Luca

Publication date:
2018

Permanent link:
https://doi.org/10.3929/ethz-b-000314408

Originally published in:
https://doi.org/10.1109/VLSI-SoC.2018.8644818

Funding acknowledgement:
162524 - MicroLearn: Micropower Deep Learning (SNF)

An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study

Pasquale Davide Schiavone*, Ernesto Sanchez†, Annachiara Ruospo†*, Francesco Minervini*,
Florian Zaruba*, Germain Haugou* and Luca Benini*
*ETH Zurich
{pschiavo@iis.ee, minervif@student, zarubaf@iis.ee, haugoug@iis.ee, lbenini@iis.ee}.ethz.ch
†Politecnico di Torino
{ernesto.sanchez, annachiara.ruospo}@polito.it

***Abstract*—The complexity and heterogeneity of digital devices used in embedded systems is increasing everyday and delivering a bug-free design is still a very complex task. The interest for open-source hardware in real products is demanding for tools and advanced methodologies for verification to provide high reliability to open and free IPs. In this work, an open-source evolutionary optimizer has been used to create functional test programs that improve the verification test set for an open-source microprocessor, enhancing in this way, the verification level of the device. The verification programs are generated to optimize code coverage metrics and are tested against a high-level model to find device incorrectnesses during the generation time. A perturbation mechanism has been included in the verification framework to cover parts of the device under verification not reachable with only software stimuli such as interrupts or memory stalls. The proposed methodology uncovered 10 bugs still present in the RTL description of the analyzed device and demonstrated the effectiveness of open-source verification tools for the next generation of open-source RISC-V microprocessors.**

1. Introduction

The increasing interest for open-source hardware is opening a new era in the silicon market. Among many, the free and open-source RISC-V *Instruction-Set Architecture* (ISA) [1] is becoming a viable option supported by industry leaders, such as Google, Micron, NXP, Microsemi, Qualcomm, Nvidia and Western Digital just to cite a fews [2]. A rich ecosystem of open-source software and hardware is growing around the RISC-V ISA. The Rocket and BOOM [3], [4] from UC Berkeley are two open-source RISC-V microprocessors already in use in companies like SiFive and Esperanto Technologies [5], [6]. The Riscy and Zero-riscy [7], [8] from ETH Zurich and University of Bologna are two more open-source RISC-V microprocessors for Internet-Of-Things (IoT) platforms used, among others, by GreenWaves Technology and Dolphin Integration. [9], [10].

Open-source cores come however with major verification challenges. Verification is already a bottleneck for the design cycle. Some surveys state that validation, verification and testing (VV&T) require about 60% [11] of the total production costs. Companies approaching open-source IPs usually verify their functionality internally and can provide reports or fixes to the IP designers to improve the quality of the free hardware. For instance, recently the *Riscy* core has been compared and chosen to be a valid candidate in an industry project by Google [12]. For that reason, it has also been extensively verified using STING [13], a versatile design verification platform. This helped in uncovering crucial bugs in the *Riscy* multiplier and the forwarding and stall logic related to the load-and-store unit. The bugs have been reported and fixed, proving the usefulness of open-source hardware in the industry context and the needs of advanced verification strategies.

However, a complete open-source verification framework for open-hardware would allow free access to verification efforts and it would increase the reliability of free IPs. Clifford Wolf from Symbiotic EDA developed an open-source end-to-end formal verification framework for RISC-V processors [14]. However, it requires changes in the DUV interfaces to be integrated in such framework. Kami is another open-source formal verification framework that has been used to verify BlueSpec written RISC-V processors [15]. On the other side, μGP (microGP) [16] is a flexible open-source evolutionary-based tool that is able to automatically generate syntactically correct assembly programs and it has been already used to test and verify microprocessors [17]. In short, the evolutionary engine receives the description of the processor assembly syntax and the evolution parameters. As a final result, the best suited test programs, or best individuals, are included in the verification suite. In most of the cases, such programs maximize the code coverage metrics of the *Device Under Verification* (DUV) and can be used to find inconsistencies between the HDL of the DUV and the trusted golden model.

In this paper, a simulation-based verification framework based on μGP is developed to increase the verification level of the *Riscy* core with automatic test program generation. Such framework relies on an evolutionary test program generator, the DUV, a golden model to evaluate the correctness of the DUV, and an independent evaluator that promotes programs that cover most of the DUV HDL description.

978-1-5386-4756-1/18/$31.00 ©2018 IEEE

The evolutionary engine drives the test program generation targeting to maximize the high-level code coverage metrics. The verification setup takes as input an instruction library that has been split and used in several evolutionary phases to optimize the final verification test set. In addition, a perturbation module has been described to produce a noisy behavior in the processor interaction with the external world (memories and interrupts). Furthermore, the generation of the programs is combined with the verification phase to leverage all the individuals created during the evolution to increase the change to uncover bugs in the HDL description. If a bug is found during the evolution, the individual can be added to the final verification suite. The proposed verification framework is composed by free and open-source program generator and DUV[1] [2] and can be adapted for others RISC-V HDL descriptions, bringing state-of-the-art verification methods to the open-hardware ecosystem. The elements used in the proposed methodology can be easily integrated in a formal verification plan as the one defined by [18].

The main contribution of the proposed methodology and case study consists on creating stimuli for the *Riscy* core that optimize a set of code coverage metrics to thoroughly explore the verification research space. In fact, the proposed setup was able to discover bugs on arithmetic functions of the core when special operands were used. For example, infinite numbers in the core *Floating-Point Unit* (FPU) or operands that cause intermediate overflows during the computation of special function in the *Arithmetic Logic Unit* (ALU).

At the end of the proposed verification process, we obtained an automatically generated verification test-set reaching on average about 90% on a set of high-level code coverage metrics, while unveiling 10 different bugs still present in the processor description.

The rest of the paper is organized as follows: Section 2 recalls some important concepts that support the paper, and summarizes some of the most important aspect about function verification for microprocessors and introduces the proposed approach, describing the main novelties of the presented strategy. Sections 3 introduces the case study and outlines the results gathered on *Riscy*. Finally, the last sections conclude the paper and draft the future works.

2. Microprocessor Functional Verification

It is possible to define functional verification as the demonstration that the intent of a design is preserved in its implementation. Many methodologies have been proposed to reach this goal targeting digital circuit verification, and can be roughly classified as static or dynamic techniques. Static methodologies (usually called *formal*) try to demonstrate that the circuit implementation conforms the specifications. Formal techniques can be classified as canonical graphical expression models, including for example BDDs [19]; and algebraic expression models, that include for example satisfiability (SAT) and integer linear programming (ILP) [20] methodologies. The main disadvantages of formal or static techniques are the huge computational resources required, even for circuits of medium complexity. Differently, dynamic methodologies (called *simulation-based*) aim at uncover design errors by exercising the actual implementation of the design. These methodologies do not suffer from the above limitations, but only cover a limited range of behaviors and will never achieve 100% confidence of correctness [21]. Interestingly, in [18], the author states that the actual success of a dynamic methodology is mainly based on a well established verification route-map. This route-map, called *verification plan*, should be composed of three main elements:

- Coverage measurement: clearly defining a verification problem, the different metrics to be used and the verification progress;
- Stimuli generation: providing the required stimulus to thoroughly exercise the devise by following the plan directives;
- Response checking: describing how to demonstrate the behavior of the device conforms the specifications.

Due to the current complexity of processor cores, a suitable solution to create a stimuli set for microprocessor verification is to resort to test or verification programs. A test program is a syntactically correct sequence of assembly instructions that is provided to the processor using the normal execution mechanisms. The test program goal is not to execute a normal task, but to try to discover any possible design or production flaw.

Since its introduction [22] in the 80's, test programs have targeted validation, verification and testing of microprocessors. One common practice for microprocessor functional verification is to resort to hand written programs made by skilled design engineers that for example exercise certain corner cases. Unfortunately, these engineers are required to have a deep knowledge of the device under verification, meaning that a considerable amount of human effort is required to create these verification programs.

A different possibility to generate functional test programs is to resort to constrained-random test generation, as proposed in [23], [24], [25]. Those techniques usually exploit program templates where some of the used values or program parameters are generated randomly, whereas others are previously defined. The main drawback of those techniques is that they are difficult to implement when targeting complex designs. A possible evolution of the previous techniques is based on the exploitation of coverage metrics able to provide feedbacks to better drive the generation process. These techniques, as described for example in [17], [26], [27], require the generation and simulation of a huge quantity of test programs, but the generation process is usually more efficient than in pure random-based approaches.

1. μGP is open-source and freely downloadable at http://ugp3.sourceforge.net/
2. *Riscy* is open-source and freely downloadable at https://github.com/pulp-platform/riscv

2.1. Proposed Approach

The framework developed in our case study is depicted in Figure 1. The setup includes a generation step (*stimuli generation*) combined with subsequent checking (*response checking*).

On the left part of the figure, it is possible to see the evolutionary optimizer called μGP [16], an evolutionary-based tool inspired by the Darwinian principle of reproduction and survival of the fittest. μGP receives the description of the processor assembly syntax trough the so called *Instruction Library*, in addition to configuration information that sets the parameters used during the evolution, for example, the number of test programs in the population, the number of instructions included on every test program, the maximum number of test programs to simulate, and so on. Then, the evolutionary process starts by creating a set of random programs, so called individuals, that are evaluated externally by a fitness function. At every evolutionary step or *generation*, the best individuals are improved using genetic operators such as *crossover* and *mutation*, while the worst ones are discarded.

The *Instruction Library* is devised targeting the specific DUV ISA. In this library, the different processor registers, instructions, rules of use, and for example, the available addressing modes are described to support the generations of test programs. Moreover, not only mere instructions are described, but more complex pieces of programs that define finite loops, illegal instructions, and special environmental parameters to throw exceptions. In addition, divisions by zero, infinite and not-a-number operations are also described to cover special cases in the microprocessor.

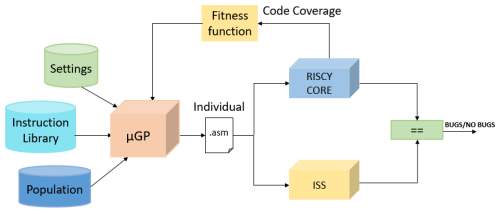

Figure 1. Verification framework

In our experiments, the verification programs are evaluated resorting to a checking scheme reported in the right part of Figure 1. In particular, the framework setup followed these steps:

1) The Instruction Library was created according to the target ISA.
2) The main μGP settings were defined.
3) A script was produced to automate the flow. Its purpose was to pick up the individuals produced by μGP generation by generation and provide it to the simulator and the ISS. Then, to create the fitness function, the code coverage percentages were collected and fed to the evolutionary algorithm again.

The fitness of every verification program is evaluated by measuring its capacity to maximize the code coverage metrics on the processor model [28] [29]. For any verification program, a fitness value was computed by resorting to an equation that includes together the different metrics; in particular, the used metrics are: *Statement Coverage*, *Branch Coverage*, *Condition Coverage*, *Expression Coverage*, *FSM State Coverage*, and *FSM Transition Coverage*.

The fitness function is computed as following: for each metric in use, the *arithmetic average*, the *variance* and the *sum* is computed accounting for all the core modules single metrics. As equation 1 highlights, this results in a vector of 18 elements which is used by μGP to drive the evolution. In particular, μGP gives more relevance to the first element of the vector fitness function and least priority to the last one. The order of elements has been decided empirically, with the first 6 elements being the average (a), the second 6 elements the variance (v) and last 6 elements the sum (s) of the *Statement Coverage* (1), *Branch Coverage* (2), *Condition Coverage* (3), *Expression Coverage* (4), *FSM State Coverage* (5), and *FSM Transition Coverage* (6).

$$fitness = \{a_1, a_2, a_3, a_4, a_5, a_6, v_1, v_2, v_3, v_4, v_5, v_6, s_1, s_2, s_3, s_4, s_5, s_6\}. \quad (1)$$

Where for instance:

$$a_1 = \frac{\text{Sum of Statement Coverage of all units}}{\text{Number of units}}$$
$$a_2 = \frac{\text{Sum of Branch Coverage of all units}}{\text{Number of units}} \quad (2)$$

In order to improve the coverage results, a perturbation module has been embedded in our framework. The perturbation module is a hardware component that generates random interrupts and introduces random stalls on both data and instruction memory interfaces. It contains memory-mapped registers to configure a stall and interrupt mode (random number of stalls/interrupts or fixed). The initialization of the perturbation module is set random before executing the generated individual. However, one can consider for future works to program the perturbation registers as part of the evolutionary process.

Together with the fitness value that is used to guide the evolutionary process, any program is also used to compare the current processor model outcome (*Riscy+FPU* in Figure 1) against a high-level and reliable model (an instruction set simulator or *ISS* in Figure 1). The ISS functional model is an accurate model of the *Riscy* ISA. For every instruction, the HDL simulator pushes the instruction word to the ISS via a DPI-C wrapper. At the end of the execution, it compares the result computed by the HDL description with the one computed by the ISS. In this way, it is possible to find differences in execution between the compared models during the evolutionary generation phase rather than using only the optimized final individuals. Once a difference is found, a report is created allowing the verification engineer to evaluate the bug of the DUV.

3. Case study and experimental results

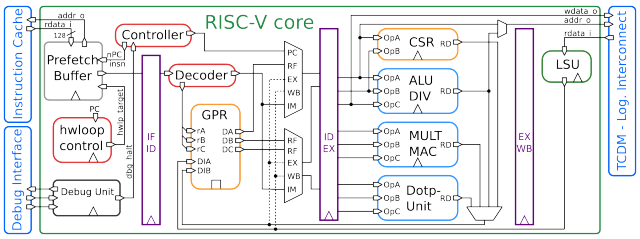

Figure 2. RI5CY Architecture

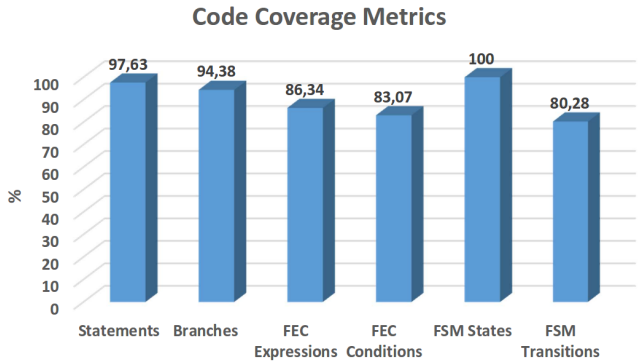

Figure 3. Code Coverage Results

To experimentally demonstrate the effectiveness of the proposed method, the RTL description of the *Riscy* microprocessor is targeted [7]. *Riscy* is an open-source, 32bit, in-order, 4 pipeline stages microprocessor based on the open-source RV32IMFC extensions of the RISC-V ISA [1] and it is described in SystemVerilog. It has been developed under the *Parallel Ultra-Low Power Platform* (PULP) project [30] and it has been extended with HW-Loop, fixed-point, bit-manipulation and *Single-Instruction Multiple-Data* (pSIMD) instructions for energy efficient computation on signal processing algorithms [7]. It occupies only 40K nand2 gate-equivalent in the umcL65 nm technology [8] plus 30K extra gates for the FPU. The core's 4 pipeline stages are the *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execution* (EX) and *Write Back* (WB) and its architecture is shown in Figure 2.

In the IF, the interaction between the core and the instruction memory bus takes place, the next program counter is calculated and the compressed instructions are decoded. The ID stage hosts the decoder, the register-file and the pipeline controller, which is also in charge to perform data-forwarding and stalling the pipeline. In the EX stage the enhanced ALU, multiplier and FPU are accommodated as well as the Control-Status (CS) register-file. The *Load-Store Unit* (LSU) sends data-memory requests during in the EX stage and receives answers in the WB stage. In addition, the WB stage takes care of two-clock-cycles FPU instructions. The instruction and data bus-interfaces implement a simple protocol via *request*, *grant* and *valid* signals. The master sends the *request* in the clock-cycle N and the arbiter can answer with the *grant* signal at clock-cycle N or later. The *valid* signal is provided by the bus-interfaces once the data from the memory is ready and it can arrive at least one clock-cycle later than the *grant* signal or later in case of multi-cycle memory accesses. *Riscy* supports up to 32 interrupts and it includes a debug-unit.

In order to verify the functionality of the DUV, it is also necessary to tests the core while handling interrupts, instruction and data bus delays (e.g., due to contention in multi-master bus or cache misses) along with the *Software* (SW) exceptions and normal instructions. Such events cannot be triggered using only instructions; therefore, external devices are involved to stimulate those conditions [31]. For this reason, a hardware implementation of the perturbation module has been designed to provide interrupt requests and bus stalls during RTL simulations. The perturbation module introduces delay-cycles both in the *grant* and *valid* signals of the core-memory interface, as well as random interrupt requests with random interrupt IDs. The memory-mapped registers allow to select the operational mode (bypass, random or not random), the number of stalls and maximum random numbers. In addition, the perturbation module can be configured to rise interrupt requests when the *Riscy* core is decoding a given instruction identified by its program counter. This mode is very useful to reproduce bugs that depend on specific conditions. The perturbation module registers are mapped in a subset of the core's debug-space so that it can be easily integrated in different platforms and its registers can be accessed by different peripherals such as JTAG or SPI or by means of load and store instructions.

The previously described framework was implemented and at the end of the experiments, the verification test set obtains a high code coverage results reported in Figure 3. It is interesting to note that the obtained results are in line with the ones obtained in [32] and described in [28]. Actually, in most of the cases, for example, in the case of the *statement coverage*, the metric saturates at the reported values. In the performed experiments, the code coverage metrics have been extracted resorting to Modelsim® HDL Simulation and used as variables of the fitness function to evaluate the program's fitness. However, any free logic simulator tool which supports code coverage estimation could be used as well (e.g. Verilator).

The *Instruction Library* contains hundreds of rules to describe the aforementioned ISA extensions and special cases, and it is split in three sub-libraries in order to allow a better exploration of the processor description and maximizing the final code coverage at the same time. The first library generates test programs that only contain RV32IMC instructions (no floats) and PULP extensions plus constraints to generate special cases as illegal instructions and functions to enter sleep mode.

To stress more complex units and corner cases, a second library has been designed to focus on FPU instructions without polluting the individuals with the generation of integer

TABLE 1. CODE COVERAGE RESULTS.

Library	Code Coverage
RV32IMFC Random	52%
Single General-Purpose	64%
Proposed Optimized Splitted	85%
Proposed Optimized Splitted + Perturbation	90%

Code coverage results for different individuals.

instructions. This approach allows to further stress the FPU with high density float instruction sequences. This library also contains special cases as RV32F illegal instructions and corner cases (NaN, infinite, division by zero). Finally, the third library has been used to generate two different experiments: one to optimize the coverage of the float multiply-and-accumulate unit and the float division-and-square root unit isolated from the rest of the DUV; and one still targeting the FPU but with focus on its conditional part. This is achieved by giving more relevance to the *Condition Coverage* metrics in the aforementioned fitness function instead of the *Statement Coverage*. The final test program generation process is split in four different experiments. The union of all the best individuals results in a final code coverage of 85%. Finally, by adding random stalls on both data and instruction memory and random interrupt requests, the final code coverage increased to more than 90%. The remaining 10% was related for example to FSM transitions which never happened, special case of NaN, infinite numbers, and their combinations. It is important to note that the main drawback when dealing with evolutionary-based tools is the simulation time. In fact, to generate one of the best individuals included in the final test set, it takes about 8 hours of simulation.

Splitting the evolutionary phases is useful to specialize the verification functions to face those parts of the DUV code difficult to cover. Indeed, a single general purpose library to generate test programs based on the RV32IMFC plus extensions (but without special conditions like explicit NaN operands or illegal instructions) was able to cover only an average of 64% of the HDL code. One of the main reasons was that the generated program did not contain enough RV32F instructions keeping the FPU coverage low. A pure random program generated by the first generation of individuals from the same library had instead only 52%. Table 1 summarizes the code coverage discussed above.

The fitness function has been designed trying to boost all the coverages metrics to grow uniformly, and if two individuals have the same coverage results, the smaller code size program is preferred.

The final test set uncovered 10 design bugs during the verification process. These errors belong to different types, for example, the computation performed by the instruction $p.clip$ and $p.extract$ were discovered to be incorrect in a very specific case. For instance, due to an intermediated overflow, the $p.clip$ instruction failed to compute the correct value with specific cases, whereas the logic shift instead of the arithmetic one was mistakenly used for the $p.extract$ instruction. Another example involves bugs related to FPU square root operations with rounding modes or with NaN operations in the RISC-V $fclass$ instruction.
The proposed method performs better than pure random and manual approaches as well as than using a single test program that covers the highest coverage as proposed in [17]. In fact, having the generation and verification phases split means using only the best individual to test the correctness of the core, whereas combining the two phases means testing all the programs generated during the evolutionary steps. This allows to explore a wider space of programs and increasing the probability of success. For example, the bug related to $p.clip$ instruction previously mentioned was not part of the highest coverage program found by the optimizer, still it has been generated among the individuals of a prior generation to be then discarded during the evolutionary process. By combining generation and verification, every individual is not only used to calculate the code coverage, but it has the possibility of being executed and compared against the golden model (the ISS) to find design bugs. This approach shows that it is not only important to maximize the code coverage but also to leverage less important individuals during the generation phase.

4. Future Work

The success of the experiment results in a more reliable verification suite to evaluate the correctness of the RI5CY core. A similar approach can be also used to verify other microprocessors of the RISC-V open-source community but it can also be adapted for other kind of HW blocks for example *Direct Memory Access* (DMA), *HW-Accelerator* (HWA) or any peripherals. Even if this methodology was followed for a RISC-V, nothing prevents it from being performed for other ISAs. Furthermore, the presented framework did not consider the verification of the core debug unit, which can be a future extension of this work. This can be accomplished by building *Instruction Libraries* that contain pseudo-random stimuli for the DUV and by adapting the perturbation module to interact with the DUV interfaces. The increasing complexity of free HW (full microcontrollers with peripherals, DMA, cluster of multicores, event-units, debug, etc) open the opportunity for verification and testing research to spread their tools over different IPs connected together.

5. Conclusion

Today's crucial role of embedded systems requires to decrease as much as possible the current time to market times; however, it is difficult to speed up the production process due to the high amount of time and resources required during the verification step. In this paper, we proposed an evolutionary-based methodology able to automatically generate assembly programs and we applied it to enhance the verification level of the RISC-V *Riscy* processor. The proposed methodology combines the use of an evolutionary optimizer, a hardware perturbation module and a checking mechanism to rapidly create verification sets of assembly

programs. The evolutionary optimizer, the perturbation module as well as the core are open-source and available for usage and further development. The experimental results demonstrated the effectiveness of the method by uncovering 10 bugs in the RTL description of the DUV.

Acknowledgment

This work has been funded by the Swiss National Science Foundation under grant 162524 (MicroLearn: Micropower Deep Learning).

References

[1] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," 2014.

[2] "Platinum members of the RISC-V foundation," https://riscv.org/membership/wpbdp_category/platinum/.

[3] Y. Lee, "RISC-V "Rocket Chip" SoC Generator in Chisel," https://riscv.org/wp-content/uploads/2015/02/riscv-rocket-chip-generator-tutorial-hpca2015.pdf.

[4] C. Celio, P. Chiu, B. Nikolic, D. Patterson, and K. Asanovi, "BOOM v2: an open-source out-of-order RISC-V core," https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.pdf.

[5] "Sifive," https://www.sifive.com/.

[6] "Esperanto technologies," https://www.esperanto.ai/.

[7] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gurkaynak, and L. Benini, "Near-threshold RISC-v core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2017.

[8] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2017 27th International Symposium on*. IEEE, 2017, pp. 1–8.

[9] "Green waves technologies," https://greenwaves-technologies.com/en/greenwaves-technologies-2-2/.

[10] "Dolphin integration," https://www.dolphin-integration.com/index.php/silicon_ip/ip_products/microcontrollers/overview/.

[11] "Sia," https://www.semiconductors.org/main/resources/.

[12] M. Cockrell. (2018) Use of RISC-V on Pixel Visual Core. [Online]. Available: https://tmt.knect365.com/risc-v-workshop-barcelona/speakers/matt-cockrell#workshop_use-of-risc-v-on-pixel-visual-core/

[13] "Running sting on pulpino platform," http://valtrix.in/programming/running-sting-on-pulpino.

[14] C. Wolf. End-to-end formal ISA verification of RISC-V processors with riscv-formal. [Online]. Available: http://www.clifford.at/papers/2017/riscv-formal/slides.pdf

[15] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala *et al.*, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, p. 24, 2017.

[16] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: the uGP toolkit*. Springer US, 2011, p. 178.

[17] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Automatic test program generation: a case study," *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 102–109, Mar 2004.

[18] P. Andrew, *Functional Verification Coverage Measurement and Analysis*. Springer US, 2008, pp. XVI, 216.

[19] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug 1986.

[20] R. Brinkmann and R. Drechsler, "Rtl-datapath verification using integer linear programming," in *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, 2002, pp. 741–746.

[21] V. Agrawal and M. Bushnell, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer US, 2002, p. 690.

[22] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Transactions on Computers*, vol. C-29, no. 6, pp. 429–441, June 1980.

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

[24] S. K. S. Hari, V. V. R. Konda, K. V, V. M. Vedula, and K. S. Maneperambil, "Automatic constraint based test generation for behavioral hdl models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 4, pp. 408–421, April 2008.

[25] Y. Zhou, T. Wang, H. Li, T. Lv, and X. Li, "Functional test generation for hard-to-reach states using path constraint solving," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 999–1011, June 2016.

[26] O. Guzey and L. C. Wang, "Coverage-directed test generation through automatic constraint extraction," in *2007 IEEE International High Level Design Validation and Test Workshop*, Nov 2007, pp. 151–158.

[27] P. H. Chang, L. C. Wang, and J. Bhadra, "A kernel-based approach for functional test program generation," in *2010 IEEE International Test Conference*, Nov 2010, pp. 1–10.

[28] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design Test of Computers*, vol. 18, no. 4, pp. 36–45, Jul 2001.

[29] E. Hung, B. Quinton, and S. J. E. Wilton, "Linking the verification and validation of complex integrated circuits through shared coverage metrics," *IEEE Design Test*, vol. 30, no. 4, pp. 8–15, Aug 2013.

[30] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu, "Energy efficient parallel computing on the PULP platform with support for OpenMP," in *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*. IEEE, dec 2014.

[31] F.-C. Yang, W.-K. Huang, J.-K. Zhong, and J. Huang, "Automatic verification of external interrupt behaviors for microprocessor design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 9, pp. 1670–1683, 2008.

[32] F. Corno, E. Sanchez, and G. Squillero, "Evolving assembly programs: how games help microprocessor validation," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 695–706, Dec 2005.