# Automatic Test Pattern Generation Using Formal Verification and Fault Injection Methods

Jad Al Halabi*†, Endri Kaja*†, Wolfgang Ecker*‡
*Email: Firstname.Lastname@infineon.com*

**Abstract**

**System on Chips (SoCs) used in safety-critical applications are required to perform at the highest levels of functional safety and adhere to reliability standards to guarantee the protection of human lives. Consequently, there is a growing need to ensure the reliability of a chip at various stages of the chip's life cycle, including in-field testing. Software-Based Self-Test (SBST) is a testing method in which the processor's available resources are employed to test the chip itself by executing a test program. In this paper, we propose an automated test pattern generation flow for SBST that combines formal verification and fault simulation to generate test patterns to test for stuck-at-0 and stuck-at-1 faults in a RISCV CPU and detect them using a fault detection mechanism based on program flow checking. Our proposed approach was successfully tested on various components of a RISCV CPU, achieving a high fault coverage and detection rate while maintaining an acceptable pattern generation time and a low area and performance overhead.**

## I. Introduction and background

The increasing complexity and size of modern microprocessors present significant challenges in ensuring their functional safety. These challenges include defects and physical damage during manufacturing, as well as the heightened sensitivity of smaller transistors to noise and environmental factors. Global safety standards such as ISO 26262 define system-level safety requirements in the automotive industry and advocate the implementation of information redundancy techniques, in addition to structural safety methods, to provide some degree of fault tolerance against unexpected system behavior. To evaluate and enhance the role of these safety mechanisms, the standard recommends the use of fault injection and fault simulation techniques to intentionally activate faults and observe the Design Under Test (DUT) in faulty operation [1].

### *A. Model-driven Architecture and Fault Injection*

To cope with the increasing design size, a high degree of automation is needed. Model-driven Architecture (MDA) has been widely explored in software and hardware development, where an abstract representation (model) is used to generate code in a target language through intermediate transformations. *Metagen* is Infineon's metamodel-based automation framework for code generation, which has improved design productivity and quality and considerably shortened lead times [2]. The system specification is captured in a *metamodel*, and Application Program Interfaces (APIs) for modifying the model are generated accordingly. Those APIs are used in Python codes to transform the model in a way that ensures conformity to the original model and hence the specification. MetaRTL [3] is a proprietary framework that generates RTL descriptions from top-level metamodel specifications. MetaRISCV [4] is an extension of the MetaRTL framework that allows developers to generate RISCV processors with different configurations by providing a means to easily instantiate and configure components developed using MetaRTL.

MetaFI [5] is another Metagen-based framework that is used to inject faults into a DUT and generate fault simulation test benches to perform different fault injection campaigns and analysis, including *Exhaustive Fault Injection* (EFI), *Statical Fault Injection* (SFI), and *Direct Fault Injection* (DFI). The fault injection technique is illustrated in Fig. 1. The *original Model of Design (MoD)* is the starting point of the fault injection flow, where it is used to generate the design RTL code with MetaRTL. The DUT intended for fault injection is selected, and a synthesis tool, e.g., Yosys [6], maps the RTL of the DUT to a Gate Level (GL) netlist using a technology library. Using the netlist as a specification, a GL *Template of Design* (ToD) is created, and the ToD is converted into an MoD representation. A series of transformations are then performed to extract the circuit's net information from the model, perform fault collapsing to optimize the number of faults, and finally add *saboteurs* to enable fault injection capabilities in the DUT. After the resulting transformed MoD (T-MoD) is obtained, it is substituted in the original design MoD to generate a system verilog description of the design. This description contains the DUT in a GL representation with fault injection capabilities and the rest of the design in a higher-level RTL abstraction. The mixed granularity design preserves the benefits of GL representation, namely the high level of observability and detail during simulation, and combats its performance drawbacks by keeping the rest of the design at a higher RTL representation. The mixed granularity design is checked for equivalence with the original design to make sure that the transformations did not affect the functionality of the DUT.

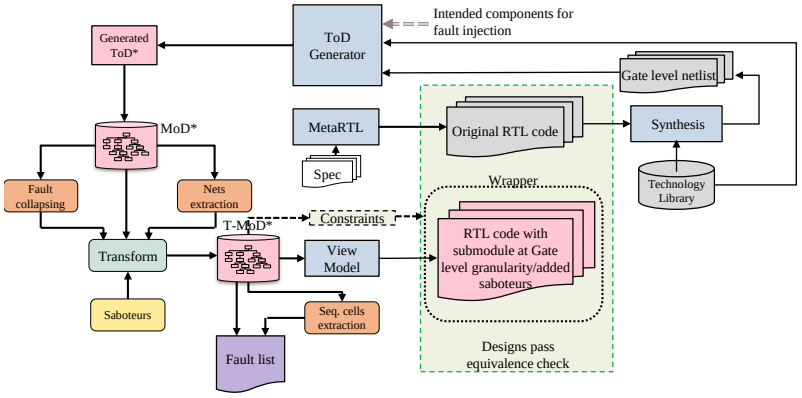


Fig. 1: Mixed register-transfer/gate level fault injector [5]

Another important aspect of design safety is testing a chip after production, with an increasing demand for in-field testing. A cost-effective and efficient approach that complies with the ISO 26262 standard is the SBST method, in which a series of instructions are executed by the chip at circuit speed to detect faults in a CPU-based design. The effectiveness of SBST is demonstrated by a *fault coverage* metric that is largely dependent on the quality of test vectors generated by Automatic Test Pattern Generation (ATPG) tools. ATPG tools struggle to generate test patterns for large sequential circuits. Scan chains are used to improve controllability during testing by providing an infrastructure to load any test vector into a DUT, which simplifies the pattern generation problem as only combinational circuits are considered. A scan chain is also used to capture the results of a test and plays an important role in a fault detection mechanism, but is expensive in terms of area, design effort, and performance.

### *B. Program Flow Checking*

To enable in-field testing, a fault detection mechanism is needed to capture and assess if faults are present when performing SBST. Many hardware faults can cause anomalies in program flow execution, potentially causing a system to malfunction or fail and raising safety concerns in an application [7]. PFC is a technique for identifying errors in program execution, such as incorrect instruction order, modified instructions, address misalignment, etc. Signatures (fingerprints) are inserted into the program during

compilation and checked at runtime. The signatures are created using a hash function such as Cyclic Redundancy Check (CRC). The process of flow checking includes creating an initial signature, updating the signature by hashing the current instruction and the previous signature, and verifying the program flow by comparing the value computed at compile time with the one computed at runtime. If a hardware fault affects the program flow, then there will be a mismatch between the two hash computations.

It is possible to calculate the runtime signature using software, but the process impairs the performance of the processor. Hardware PFC modules, on the other hand, calculate the runtime signature by implementing the hash function in hardware and storing the hashed signature in a register that can be accessed by software.

In this paper, PFC units are used as fault detection mechanisms to detect faults that go beyond the execution of the program flow and include faults that propagate to key observation points by hashing signal values. The fault detection mechanism complements a SBST generation technique that leverages formal property checking and fault simulation guided by MetaFI to detect faults in a RISCV CPU generated with MetaRISCV.

## II. Related work

The use of boolean differences in Test Pattern Generation (TPG) tools was initially disfavored due to the tedious manipulation and solution of boolean difference formulas [8]. However, practical techniques have been developed to address this issue. For example, Gurumurthy et al. [9] expressed the signal propagation requirements as a Linear Temporal Logic (LTL) formula [10] and utilized a mature bounded model checker to solve the problem [11]. Additionally, the work demonstrated a fault coverage of over 90% of functionally testable faults for various components [12]. Another approach by Lingappan et al. [13] involved the use of fault simulations to determine the number of detectable faults by a generated instruction sequence and adjust the solver settings to optimize generation time and fault coverage.

Furthermore, Riefert et al. [14] employed an unbounded model checker as a TPG tool and demonstrated a high fault coverage, proving the untestability of faults and improving runtime using heuristics. They also introduced a Validity Checker Module (VCM) to constrain a design written in HDL, which was later applied by Faller et al. [15] with the addition of a higher abstraction layer for reusability of constraints for different processor cores. These approaches involved the synthesis of the VCM and a miter circuit into a gate-level representation, leading to the observation of the DUT state in the presence and absence of faults.

On the other hand, researchers such as Chen et al. [16], Paschalis et al. [17], Kranitis et al. [18], and Gizopoulos et al. [19] have introduced various techniques for structural and software-based testing of processor components, focusing on enhancing fault coverage and test development efficiency.

## III. Pattern generation

A stuck-at-0 or stuck-at-1 fault can affect a circuit's functionality if certain circumstances *activate* the fault and *propagate* its effect to a primary output. The idea behind the technique presented in this paper is to use a formal tool to check whether a fault affects the functionality of a DUT.

A *miter* setup is created to allow the concurrent comparison of a DUT in normal and faulty conditions. By formulating a property as shown in Fig. 2, a fault can be injected using the *assume* part of the property and the *prove* part of the property to assert that the outputs of the two DUTs are identical.

If the injected fault affects the functionality of the design, then the formal tool will find a *counterexample* to *disprove* the assertion and in doing so, provides the sequence of inputs that lead to the fault *activation* and *propagation*.

```
property generate_pattern;
@( posedge clk ) disable iff ( rst )
    active_fault |-> Normal_DUT.output == Faulty_DUT.output;
endproperty
```

Fig. 2: TPG property

To apply this concept to the RISCV CPU, the generation process needs to be guided to produce a complete SBST program that is composed of *legal instructions* and can be executed sequentially without skipping any instruction. These constraints are given in the form of *assume* properties to:

- Constrain the instruction input space to a set of legal instruction according to the CPU Instruction Set Architecture (ISA)
- Disable interrupts
- Constrain the program counter to increase linearly therefor to disable branch and jump instructions
- Inject exactly 1 fault which remains active throughout the checking process
- Disable Control and Status Register (CSR) instructions

## IV. Fault detection mechanism

A PFC based fault detection mechanism is described using MetaRTL, the block diagram of the hardware is illustrated in Fig. 3. A CRC hashing function is implemented in hardware according to a CRC polynomial whose parameters are defined within the RISCV generation framework. The function hashes runtime signal values (hash in) with the existing value in the PFC state register, and stores the runtime signature in the same register.

The RISCV generation framework MetaRISCV can automatically instantiate the hardware in different locations in the CPU to hash different signals. The framework automatically introduces new CSR addresses for customized operations to start, stop and read the runtime hash value.

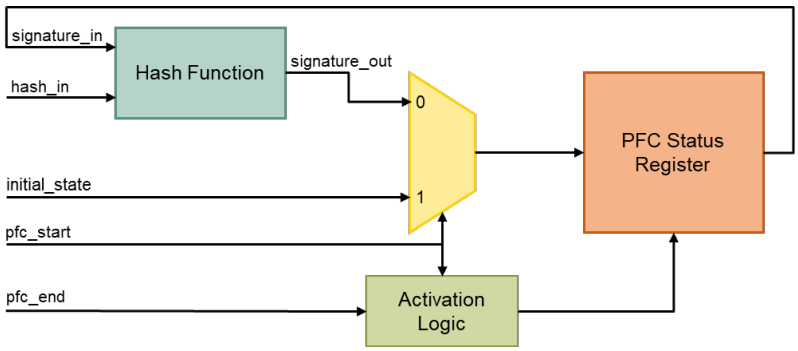


Fig. 3: PFC hardware

## V. SBST flow

The scalability of our SBST method is supported through the development of a fully automated flow encompassing hardware setup, test pattern generation, test pattern concatenation, and fault detection. Fig. 4 provides an overview of the SBST generation flow.

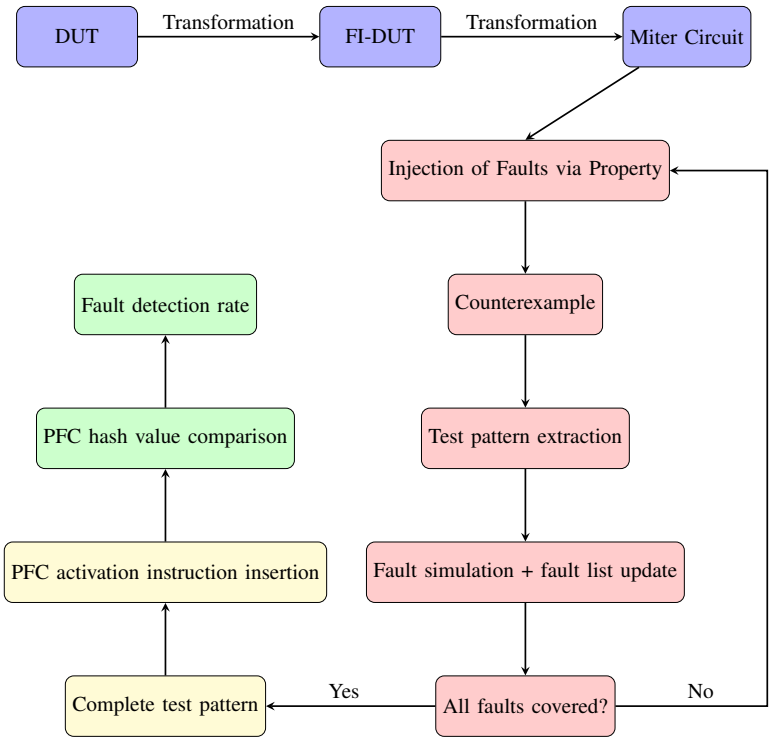


Fig. 4: SBST flow overview

The DUT, in this case, the RISCV CPU, undergoes a fault injection transformation through MetaFI. Subsequently, a complete fault list and a fault simulation test bench are automatically generated. The Fault-Injected DUT (FI-DUT) is automatically instantiated with the original DUT in a wrapper where their inputs are connected thereby creating a miter circuit ready to be loaded into the formal tool. The FI-DUT is also loaded into a simulator along with the generated test bench.

The fault control signals are encoded in a macro for ease of automation, and a script generates a property file that activates a single fault chosen from the fault list in the FI-DUT. Property checking is then initiated by the script, and if a counterexample is found, the instructions are extracted and saved. An exhaustive fault simulation is then started, and the generated pattern is read into the simulation. The results of the fault simulation which contains fault propagation information, are logged into a file that the script scans to classify the detectable faults. The detected faults are removed from the fault list, and a new fault is chosen from the remaining set. If a counterexample is not found within a selectable time period, the fault is deemed undetectable and also dropped from the fault list. This process iterates until all faults in the design are classified as detectable or undetectable. The checking timeout is important to prevent long property checking times to keep the TPG time reasonable while maintaining a good fault coverage.

The final SBST program for testing for all faults in the DUT is created by concatenating the extracted individual test patterns. Since the formal tool assumes a *reset state* as an initial state for property checking, then it is necessary during the test to reset different memory elements in the CPU such as registers to prevent fault masking. The register file of the CPU is modified to allow the reset of its content using

a custom CSR. A similar effect can be achieved with the pipeline registers by executing a number of No Operation instructions (NOPs) to *flush* their content. As a result, the automation script concatenates the test patterns by adding the created register file reset instruction and 5 NOP instructions before the execution of each pattern as shown in Fig. 5

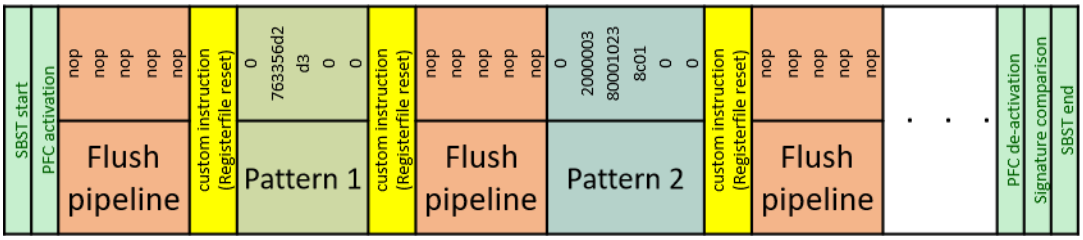


Fig. 5: Pattern concatenation example

As a feature of MetaRISCV, it is possible to attach a hardware PFC component to any signal in the design. PFC activation instructions are automatically added to the top of the SBST instruction chain and a fault simulation is performed to calculate the expected PFC hash value for each added PFC. Deactivation and comparison instructions are added at the end of the SBST program to compare the expected PFC hash with the actual runtime value.

The PFC detection rate is finally obtained by performing an exhaustive fault simulation of the complete SBST with the activated fault detection mechanism.

## VI. Application and Results

The pattern generation flow and the PFC fault detection mechanism were applied to various components of an RV32IMC RISCV CPU to evaluate the effectiveness of this methodology. The tested components include combinational components such as the ALU, HDU, and Decoder, as well as sequential components such as the register file and the ME_WB pipeline register. The formal tool used is OneSpin 360 DV-Verify® by SiemensEDA, which is run on an 11th Gen Intel® Core™ i7-1185G7 @3.0GHz processor with 32 GB of RAM. Fault simulation is performed using the Xcelium® simulator. The primary development environment for this process is a 64-bit version of RHEL 7.

The results of the SBST flow for different components in the RISCV CPU are tabulated in Table I.

TABLE I: Test pattern generation flow results

| Component | Results | | | | | |
|---|---|---|---|---|---|---|
| | # Fault | # Patterns | # Instructions | DF(%) | PFC DR(%) | Generation time (h) |
| HDU | 111 | 19 | 266 | 91.89 | 100 | 0.61 |
| ALU | 4732 | 313 | 4382 | 97.63 | 100 | 40.5 |
| Register File | 11747 | 492 | 6888 | 96.86 [1] | 100 | 13.6 |
| Decoder | 3224 | 104 | 1456 | 76.3 | 92.19 | 38.87 |
| MEM-WB | 410 | 10 | 140 | 54.8 | 100 | 8.5 |

### A. HDU

The Hazard Detection Unit is an example of a small sized component with 111 faults. The flow produced 19 patterns to detect 91.89% of the total faults in about 37 minutes.

### B. ALU

The flow generated 313 patterns that successfully identify 4620 faults in the ALU within about 13.6 hours. However, 112 faults remained undetected and were aborted after exceeding a 2-minute limit set for each test. This results in a fault coverage of 97.63%. The PFC effectively monitors the output of the ALU and detects all faults by hashing the results and comparing them to an expected fault-free reference value.

### C. Register File

The SBST flow generated 492 patterns that successfully identify 11379 of 11747 faults over a period of approximately 40.5 hours. Only 368 faults were aborted and therefore considered unresolved. This results in a fault coverage of 96.87%.
Since two PFC modules monitored the outputs of the register file, all faults can effectively be caught.

### D. Decoder

The decoder has 3224 faults. According to the experimental results of previous work [20], the decoder contains about 22.02% redundant faults, which was due to a flaw in the generation framework. Therefore, it is expected that TPG would take more time since an undetectable fault would need to be checked by the property for the entire timeout period. According to some calculations, the number of undetectable faults are around 710 and if the property check is set to 2 minutes, the flow would take 23.7 hours in efforts to find undetectable faults. The total generation time is about 39 hours, suggesting that the process can be more efficient. The fault coverage is consistent with the aforementioned work, with a value of 76.3%.
The detection mechanism was also able to detect about 92.19% of the faults, while the others were unlikely to affect the CPU outputs as the faults do not affect the values in the register file nor the ALU.

### E. Memory Writeback Pipeline Register

The Mem-Wb pipeline register had 410 faults. The flow generated 10 patterns to test 225 functional faults in 8.5 hours of generation time. The lower fault coverage is attributed to an increase in the checking complexity. The ALU PFC was able to detect all the testable faults.

## VII. Conclusion and future work

In this paper, we have presented an automated method for creating SBST for RISCV processors developed with MetaRTL. The proposed method employs formal verification techniques to generate test patterns, resulting in a high fault coverage of over 91% for most tested components. By combining formal verification and fault simulation, we conduct fault dropping to minimize the number of required properties. Additionally, the SBST generation flow integrates a PFC module, ensuring a fault detection rate of 100% for most components, aligning with various ASILs of the ISO 26262 standard. This technique operates fully automatically using MDA principles.
**Future work:** We aim to extend the application of the SBST technique beyond RISCV processors to assess its adaptability across different processor architectures. Furthermore, the flow has some limitations since it does not guarantee the undetectability of faults. The drawbacks were mitigated by introducing a generation time limit that balances generation time and fault coverage. Our future work involves conducting additional experiments on various processor components, refining test patterns by integrating manual code for components that are easier to test and reduce the property checking complexity by enabling arbitrary register writes.

## REFERENCES


[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166 – 182, Feb 1990.

[2] W. Ecker, M. Velten, L. Zafari, and A. Goyal, "The metamodeling approach to system level synthesis," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–2.

[3] J. Zhu, "Metartl: raising the abstraction level of rtl design," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001, pp. 71–76.

[4] K. Devarajegowda, E. Kaja, S. Prebeck, and W. Ecker, "Isa modeling with trace notation for context free property generation," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 619–624.

[5] E. Kaja, N. Gerlin, M. Vaddeboina, L. Rivas, S. Prebeck, Z. Han, K. Devarajegowda, and W. Ecker, "Towards fault simulation at mixed register-transfer/gate-level models," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6.

[6] Yosys, "Yosys – Open SYnthesis Suite," https://yosyshq.net/yosys/, [Online: accessed 01-November-2024].

[7] N. O. Leon, "Software Flow Monitoring using Model-Driven Development," Master's thesis, Hochschule Darmstadt, 31,12,2020.

[8] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, 1992.

[9] S. Gurumurthy, S. Vasudevan, and J. A. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," *2006 IEEE International Test Conference*, pp. 1–9, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:14907966

[10] Z. Manna and A. Pnueli, "The temporal logic of reactive and concurrent systems," in *Springer: New York*, 1991. [Online]. Available: https://api.semanticscholar.org/CorpusID:34600346

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:524729

[12] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," *Proceedings. International Test Conference*, pp. 590–598, 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:316452

[13] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 5, pp. 518–530, 2007.

[14] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker, "A flexible framework for the automatic generation of sbst programs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3055–3066, 2016.

[15] T. Faller, N. I. Deligiannis, M. Schwörer, M. S. Reorda, and B. Becker, "Constraint-based automatic sbst generation for risc-v processor families," in *2023 IEEE European Test Symposium (ETS)*, 2023, pp. 1–6.

[16] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369–380, 2001.

[17] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001, pp. 92–96.

[18] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, 2005.

[19] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, pp. 1441 – 1453, 12 2008.

[20] B. Zhao, "Automated Safety Mechanism Verification Technique guided by Processor Formal Verification Methods," Master's thesis, Technical University of Kaiserslautern, 01.02.2023.