A Statistical and Model-Driven Approach for Comprehensive Fault Propagation Analysis of RISC-V Variants

Endri Kaja*†, Nicolas Gerlin*†, Ungsang Yun*‡, Jad Al Halabi*†, Sebastian Prebeck*‡,
Dominik Stoffel †, Wolfgang Kunz †, Wolfgang Ecker*‡
Am Campeon 1-15, Neubiberg - 85579, Germany
*Infineon Technologies AG
‡ Technische Universität München, Germany
Email: Firstname.Lastname@infineon.com
†Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau
Email: dominik.stoffel@rptu.de, wolfgang.kunz@rptu.de

Abstract

The increasing complexity of modern designs, combined with strict safety standards, presents new challenges in ensuring the reliability of systems. Developing effective safety verification techniques is crucial to prevent any safety-related errors from entering the production phase. Although design complexity poses a significant challenge in both the design and verification processes, it is important to acknowledge that safety verification is inherently difficult and presents various challenges, independent of design complexity. The ISO26262 guidelines recommend using fault injection to verify safety-critical designs, but this process is time-consuming and labor-intensive. In this paper, we propose an automated and model-driven statistical fault injection technique that can be applied to different design variations. The Statistical Fault Injection (SFI) approach reduces the number of injectable faults, thereby reducing overall complexity. We applied the technique to two RISC-V variants while running four different CPU workloads. The experimental results demonstrate the effectiveness of SFI, providing reliable results with a high level of confidence.

I. Introduction and Background

The rapid and exponential advancement of technologies in recent years, including AI accelerators, the RISC-V ecosystem, the Internet-of-Things (IoT), Cyber-Physical Systems, and Smart Systems, among others, has led to a complex development process. This process requires not only intelligent and automated digital design techniques, but also sophisticated methods to ensure the accuracy of these designs. However, despite these technological advancements, the verification process remains a significant challenge. It acts as a bottleneck, hindering design productivity and compressing Time-to-Market schedules. The increasing complexity of IC/ASIC design and verification can be attributed to multiple factors, including the growing size of designs and the emergence of additional design prerequisites. These prerequisites extend beyond basic functionality and encompass elements such as safety and security. Many System-on-Chips (SoCs) are now utilized in safety-critical domains, such as automotive, aerospace, healthcare, and financial systems. In these domains, the utmost priority is given to developing systems or products that prioritize safety above all else. These specialized systems are known as safety-critical designs, as they play a critical role in safeguarding human lives and maintaining the integrity of sensitive operations. According to Wilson Research Group, 44% of ASIC projects have integrated safety-critical features into their designs [1]. The international automotive standard ISO26262 recommends the use of *fault injection* to verify the safety-critical aspects of designs.

The process of fault injection plays a critical role in evaluating the resilience and fault tolerance of a system. By purposely introducing faults, designers can gain valuable insights into how the system behaves under different scenarios. When dealing with designs that consist of N nodes, there is an exponential number of possible single stuck-at fault models, amounting to 2N possible faults. Given the arbitrary nature of faults, fault injection campaigns typically adopt a statistical and probabilistic approach, such as Statistical Fault Injection (SFI). SFI involves selecting a random subset of faults from the complete set and subjecting them to simulation. The sample coverage, which represents the ratio of detected faults in the random sample to all faults in the sample, is then derived. This coverage is used to estimate the fault coverage in the complete fault set. To complete the process, the margin of error and confidence interval are evaluated. The objective of SFI is to reduce the computational effort required for fault injection while still obtaining a reliable estimation of the system's fault coverage. To effectively meet the various requirements of safety analysis, an automated and efficient fault injection tool becomes indispensable. This tool serves as a bridge between the efforts invested and the final fault injection campaign, ensuring that the process is streamlined and meets the necessary standards.

In this paper, we introduce a versatile and automated framework that is designed to generate a wide range of SFI campaigns, while also bridging the gap between specifications and the fault injection process with minimal effort. The framework is vendor-independent, allowing it to be used with any Verilog/SystemVerilog-based simulators/emulators. Ecker et al. [2], [3] developed a proprietary automated RTL generation framework called MetaRTL, which utilizes Python APIs to generate RTL (Register Transfer Level) designs based on top-level metamodel specifications. By using this framework, manual efforts are significantly reduced, leading to increased design productivity. The RTL generation flow is employed to generate designs with a combination of RTL and gate-level granularity, while also equipping them with fault injection capabilities [4], as presented in Figure 1.

In the fault injection process, the design RTL is first created using MetaRTL and then synthesized to generate the gate-level netlist. To select the intended component for fault injection, the Template-of-Design (ToD) Generator, a Python script, reads both the netlist and user input. Using this data, the ToD Generator produces a new ToD with mixed granularity. For instance, if the Execute stage of a pipelined processor core is chosen for fault injection, the resulting ToD only defines the Execute stage using gate-level granularity based on the netlist data. The rest of the processor's components are represented identically to the original RTL. Afterward, the Model-of-Design (MoD) is modified by adding saboteurs, which are small hardware components that can inject a fault when activated. Only the selected parts of the design that are subjected to fault injection are kept at the gate-level granularity. Meanwhile, the remaining parts are represented at the RTL granularity. This approach enables fast RTL fault simulation without compromising the accuracy of gate-level fault simulation. Consequently, it allows for automatic SFI on different RISC-V variants.

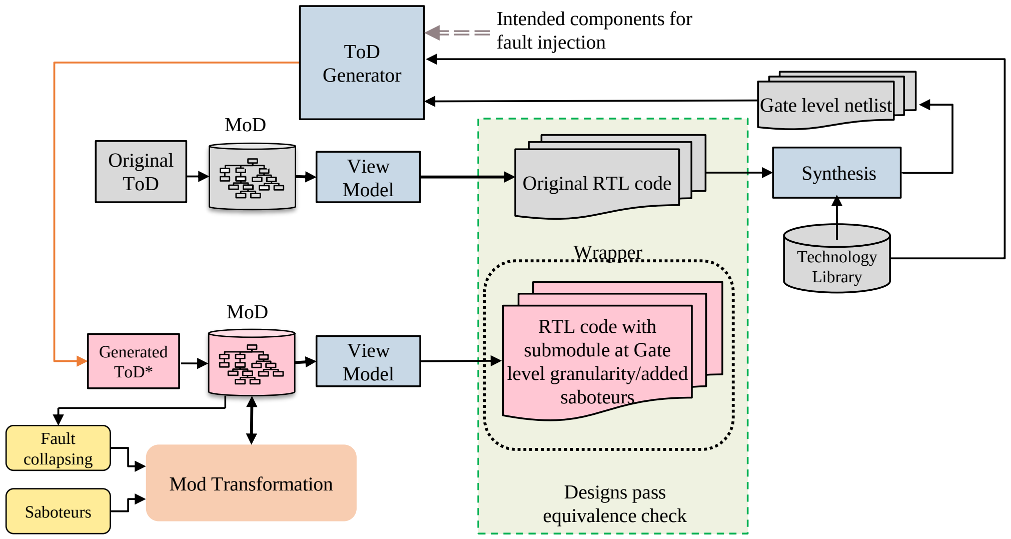


Fig. 1: Mixed register-transfer/gate level fault injector

II. Related Work

Fault injection and safety verification is a well-researched topic in academia as well as industry. In the following, some of the related techniques are presented.

Cadence® Jasper™ FSV tool facilitates formal fault injection and propagation analysis for supporting safety and security verification. Similarly, Siemens EDA ® offers formal verification methods, such as Siemens Architectural Analysis and Siemens Analyze Faults, which allow users to define custom fault scenarios or choose from pre-defined ones. It's important to note that these tools are vendor-specific. In contrast, GenFi [5] is an academic approach that involves injecting faults on microarchitectural simulators like MARSS and Gem5. Baraza et al.'s [6], [7] method, on the other hand, involves modifying the hardware description of the design using saboteurs and mutants. In another work, Kammler et al. [8] propose a Verilog-based framework that utilizes the Verilog Programming Interface (VPI) for performing fault injection. Kaliorakis et al. [9] have developed and introduced a methodology for incorporating fault injection capabilities into microarchitectural simulators. Specifically, they have enhanced existing simulators (MARSS and Gem5) to include fault injection capabilities, called MaFIN and GeFIN respectively. They have also implemented a Fault Mask Generator that generates random fault masks for various types of faults, such as bitflips, permanent faults, and intermittent faults. These fault masks are then processed by an Injection Campaign Controller, which sends injection requests to the Injector Dispatcher, a module that interfaces directly with the simulators. The proposed methodology was evaluated through experiments on x86 and ARM processors. In a similar vein, another study by Parasyris et al. [10] modifies the Gem5 simulator to introduce additional functionalities to threads, allowing for the modification of signal run-values.

III. Statistical Fault Injection

The SFI campaign, which aligns with ISO26262 standards, is widely recognized as the most commonly utilized technique. To comprehend the fundamental mathematical definitions of a SFI campaign, it is important to consider fault sampling. The accuracy of estimating fault coverage is dependent on the sample size, which represents the total number of faults in the sample. A larger sample size reduces the margin of error in the coverage estimation. Determining the appropriate sample size is based on the desired level of accuracy for the coverage estimation. Leveugle et al. [11] propose methods for estimating and quantifying the error associated with statistical fault injection, offering valuable insights for confidence in the fault injection outcome. Their main hypothesis assumes that the properties of the entire fault injection population, including all potential fault models occurring at any clock cycle, adhere to a normal distribution. The process of random fault sampling ensures that each individual fault configuration within the initial population, at a specific cycle, has an equal likelihood of being selected for the sample. This is achieved by using a uniform distribution for the random selection [11]. Thus the amount of n faults to inject per campaign can be calculated by [11]:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{Z^2 \times p \times (1-p)}} \tag{1}$$

where:

- N is initial population size, i.e. all possible faults to inject at every clock cycle,
- p is the standard error and proven that should be 0.5,
- e is the margin of error, and
- Z is confidence level parameter, i.e. the probability that the exact value is actually within the error interval.

Typically N is very large, thus the equation 1 can be simplified by assuming $N- > \infty$ as follows:

$$n = \lim_{N \to \infty} f(N) = \frac{Z^2}{e^2} \times p \times (1-p) \tag{2}$$

As an example, let us consider a design that has 10^4 fault locations and simulation will run for 10^6 clock cycles. N is calculated as 10^{10} and can be assumed as very large, i.e. ∞. The amount of faults to inject with given error margin and confidence level is shown in Table I [11]. As can be seen from the table, even for very large fault injection population, only 23784 fault injections are required to achieve 99.8% confidence of fault injection results with 1 % error margin.

TABLE I: Total amount of fault injections according to error margin and confidence level

	95% confidence z = 1.96	99%confidence z = 2.5758	99.8% confidence z = 3.0902
e = 5%	n = 384	n = 663	n = 955
e = 1%	n = 9581	n = 16519	n = 23874

IV. Model-driven statistical approach for fault propagation analysis

The paper presents a comprehensive and adaptable framework that focuses on generating a wide range of statistical fault injection campaigns while minimizing the effort required to bridge the gap between specifications and the fault injection process. The framework is designed to be independent of any specific vendor, making it compatible with various Verilog/SystemVerilog-based simulators/emulators. The framework begins with a reader component, responsible for reading the design and converting the fault list into a formal model. Once the fault list is formalized, the next step involves using an intermediate model called the FI-model to define the fault injection features. This model includes all the necessary information for conducting the fault injection campaign, such as fault attributes and design strobes (signals) to be analyzed. Finally, a template engine is employed to translate the model into a final fault handler testbench, which can be implemented in either SystemVerilog or Verilator-based C++ [12]. This allows for flexibility and compatibility with different simulation environments.

The core element of the automated injection campaign is the MetaFI metamodel (FI stands for Fault Injection) that specifies the type of the fault injection campaign as well as the certain outputs/registers that should be analyzed during the campaign. This metamodel is depicted in Figure 2. In a more comprehensive and scientific manner, the metamodel can be described as a framework that outlines all the essential components and characteristics required for a fault injection campaign. These components include: (i) *Fault Controller*, (ii) *Fault List*, and (iii) *Fault Analyzer*. The metamodel's root node establishes connections with various sets of classes based on these aforementioned components. The *Fault Controller*, which forms a part of the metamodel, consists of a collection of classes and attributes designed to define the purpose of fault injection and fault models. Within the *Fault Controller* class, there are three key attributes: *TopModule*, which identifies the highest-level module

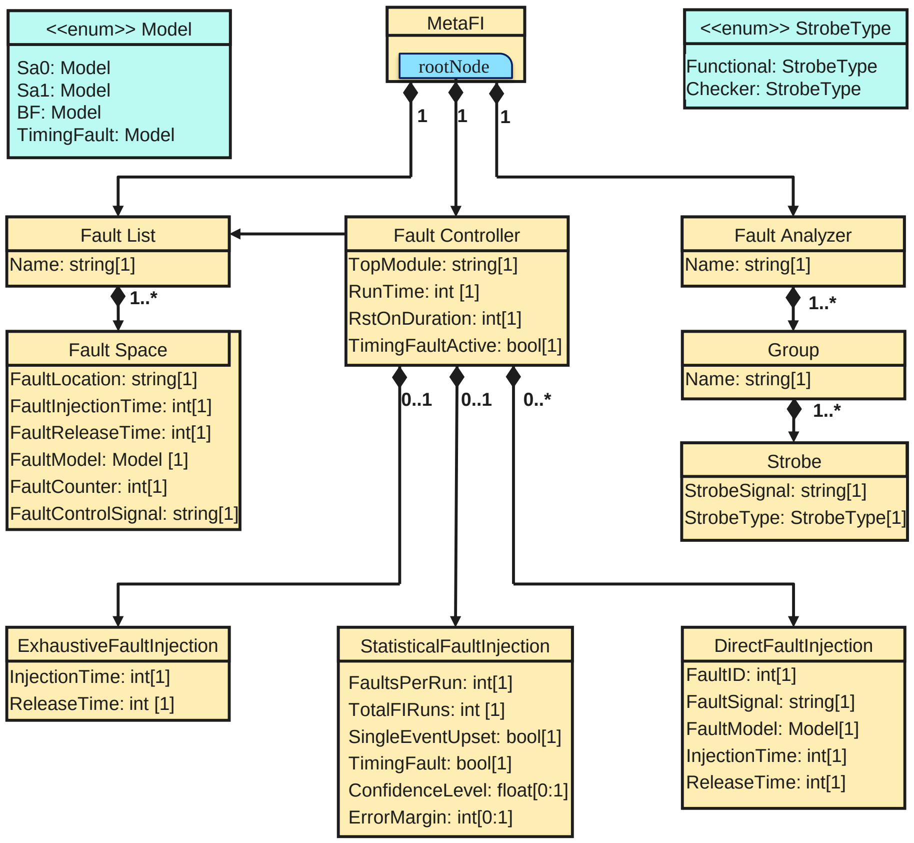


Fig. 2: Fault injection metamodel

of the RTL design for conducting fault injection; *RunTime*, which determines the duration of the simulation or emulation run in terms of the number of clock cycles; *RstOnDuration*, which specifies the duration of the active reset; and *TimingFaultActive*, which indicates whether timing faults are supported. Furthermore, the Simulation Controller class is associated with three other classes, namely *StatisticalFaultInjection*, *DirectFaultInjection*, and *ExhaustiveFaultInjection*, through a composition relationship. However, for the purposes of this paper, only the features related to SFI are considered. The *StatisticalFaultInjection* class within the metamodel possesses the following attributes: *FaultsPerRun*, which represents the number of faults to be injected in a single run, indicating the multiplicity of faults; *TotalFIRuns*, which defines the number of independent fault injection runs to be performed; *SingleEventUpset*, which determines whether only bit flips at memory cells should be introduced; *TimingFault*, which determines whether only timing (delay) faults should be injected; *ConfidenceLevel*, which represents the level of confidence associated with the sampling process; and *ErrorMargin*, which denotes the acceptable margin of error, both of which can be calculated based on the *TotalFIRuns*.

The *Fault List* class is responsible for documenting the details of faults that have been deliberately introduced into a system, also known as the fault space. This class has a one-to-many relationship (1..*) with the *Fault Space* class, meaning that each fault within the fault space is considered for fault injection. The fault space encompasses all the characteristics and properties of the injected faults. These attributes include the *FaultLocation*, which specifies where the fault is located within the system; the *FaultInjectionTime*, which indicates the time at which the fault is injected into the system; the *FaultReleaseTime*, which denotes the time at which the fault is removed from the system; the *FaultModel*, which describes the specific model or type of fault being injected; the *FaultCounter*, which keeps track of the number of times the fault injection has been executed; and the *FaultControlSignal*, which determines the signal that controls the occurrence of the fault at the specific fault location.

The *Fault Analyzer* is a component of the metamodel that is responsible for collecting and evaluating data obtained from fault injection experiments. This component has a one-to-many relationship with the *Group* class, which represents various modules and submodules within the design, and their outputs that are intended for analysis. The *Group* class establishes a one-to-many relationship with the *Strobe* class, which defines all the output signals that need to be analyzed. The *StrobeSignal* attribute is used to identify a specific output signal. Furthermore, the *StrobeType* attribute is used to differentiate between a functional strobe, which represents the functional output of the design, and a checker strobe, which represents the output of a safety mechanism. The enumeration class *StrobeType* provides a categorization of the different types of outputs.

A Python script extracts data from a metamodel instance. This data is then used to generate testbench code in both SystemVerilog and C++. Using the generated testbench, the DUT is created using MetaRTL. Subsequently, the fault simulation framework is employed to execute a fault simulation campaign. During the faulty simulations, the resulting data is logged for each simulation. This logged data is then compared to the data obtained from a non-faulty simulation, which serves as a reference. This comparison allows for the classification of faults based on their impact on the design's behavior. Finally, the results of the fault classification process are stored in a log file for further analysis.

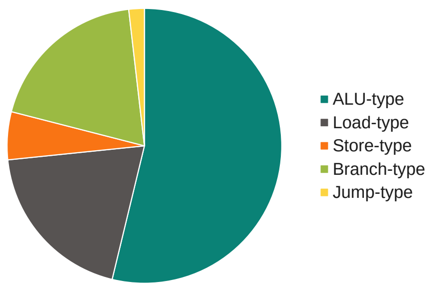
V. CPU WORKLOADS FOR FAULT PROPAGATION ANALYSIS

We selected four distinctive benchmarks to serve as workloads for the CPU, aiming to simulate realistic fault propagation scenarios. These carefully chosen benchmarks encompass a range of computational domains and offer a comprehensive assessment of the system's behavior under various conditions. The primary objective of our selection is to create a robust framework for fault propagation analysis, and each benchmark is chosen for its specific characteristics and relevance.

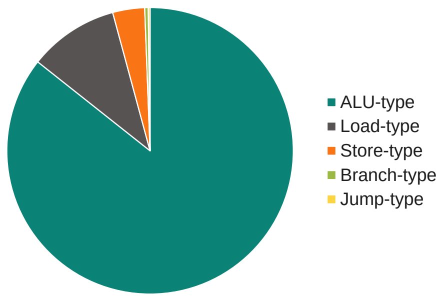
The first benchmark in our lineup is the well-known Dhrystone benchmark, originally developed by Reinhold P. Weicker in 1984. Dhrystone has historically been used as a synthetic benchmark for evaluating the performance of system integer programming. While it is true that the benchmark's structure is considered relatively simple by today's standards, its enduring fame and the high percentage of control flow operations it involves make it an intriguing workload for our fault propagation analysis. Despite its somewhat antiquated nature, Dhrystone's continued significance in the field warrants its inclusion.

The remaining three benchmarks we incorporated into our study are cryptographic algorithms, each serving a unique role in our fault propagation analysis. These benchmarks are SHA-256, MD5, and CRC-32, which represent cryptographic hash functions and checksum algorithms. These choices are drawn from the Embench suite (Embedded Benchmark), an open-source collection of C benchmarks consisting of 22 real-world applications typically deployed on deeply embedded systems. Fig. 3 displays the instruction statistics of the aforementioned benchmarks, i.e. the percentage of different instruction types.

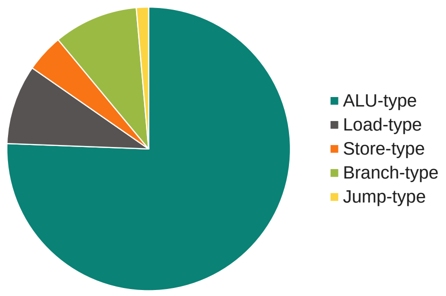
We specifically opted for cryptographic algorithms within the Embench suite due to their widespread utilization and the high stakes associated with security implications in the event of faulty behaviors. The inclusion of cryptographic workloads in our analysis is not only pertinent but also necessary, given the critical role they play in ensuring data integrity and confidentiality. These algorithms pose distinct challenges in terms of fault tolerance and error propagation, making them valuable additions to our suite of benchmarks for fault analysis. All benchmarks employed in our study were compiled using GCC 11.1, with optimization level -O3. Additionally, we configured the newlib 4.1.0 library with optimization level -O2 and garbage collection.



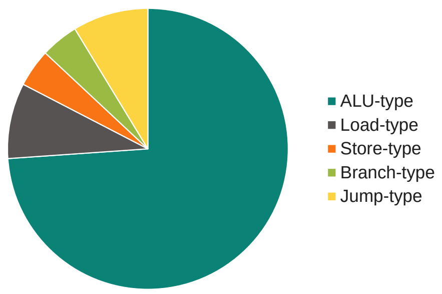
(a) Dhrystone instruction statistics



(b) SHA-256 instruction statistics



(c) MD5 instruction statistics



(d) CRC-32 instruction statistics

Fig. 3: Instruction statistics for different benchmarks

VI. Application and evaluation

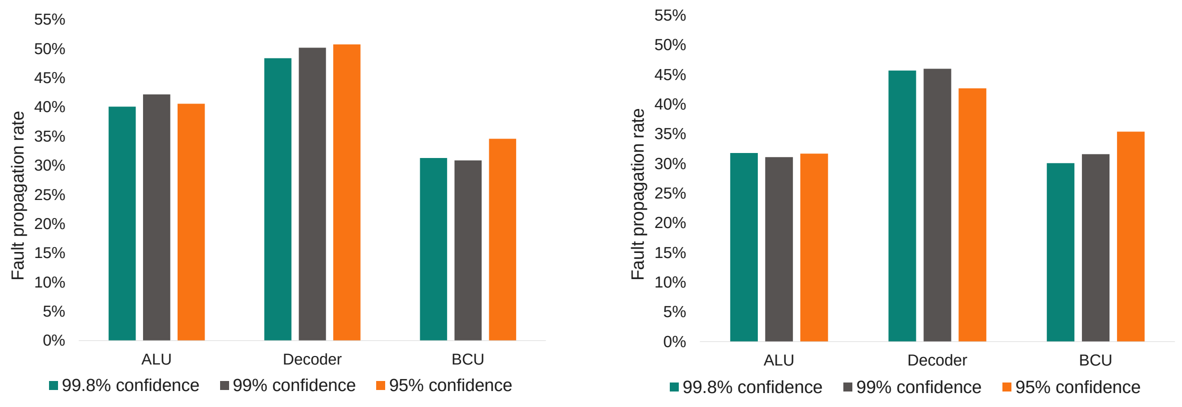
In order to implement and evaluate our approach, we utilized MetaRTL to create two distinct RISC-V-based CPU subsystems. One subsystem consisted of a 2-stage pipelined CPU, while the other consisted of a 5-stage pipelined CPU. These Designs-under-Test (DUTs) included components such as the RV32-IMC CPU, Instruction Memory and Data Memory, Instruction and Data Bus, and Peripherals. The specifications of the CPUs were defined using a metamodel, allowing for the generation of different RISC-V variants by adjusting parameters like supported instructions and extensions. Our fault injection experiments targeted specific components like the Arithmetic-Logic Unit (ALU), Instruction Decoder, and Branch Control Unit for the purpose of SFI. These components were described at a gate-level granularity, while the rest of the DUT remained at the original RTL granularity. Random fault models were injected at random locations during random simulation times, as SFI is applicable to random faults. The framework automatically determined the quantity of faults to inject based on the desired level of confidence and error margin, as illustrated on Table I. The only inputs required from the user were the simulation duration, level of confidence, and error margin. Figures 4-7 depict the Fault Propagation Rate, which refers to the proportion of faults that propagate to the primary outputs of the CPU, during the execution of SFI on the three mentioned components. The SFI process was carried out with a 5% error margin and three varying confidence rates: 95% (with 384 injected faults), 99% (with 663 injected faults), and 99.8% (with 955 injected faults). The simulation was conducted over a total of 37868 clock cycles. Table II showcases the total number of fault locations for each of the three components, namely the ALU, Decoder, and Branch Control Unit. As observed from the table, the ALU of the 5-stage CPU exhibits 8.1% more fault locations compared to the 2-stage CPU, while there is only a slight increase in the number of locations for the 5-stage CPU Decoder. Conversely, there is no change in the number of locations for the Branch Control Unit, because the change on the design does not affect the way how branching is handled.

TABLE II: Total amount of fault locations

Component	2-stage pipelined CPU	5-stage pipelined CPU
ALU	5376	5812
Decoder	3504	3514
Branch Control Unit	754	754

A. General Observations

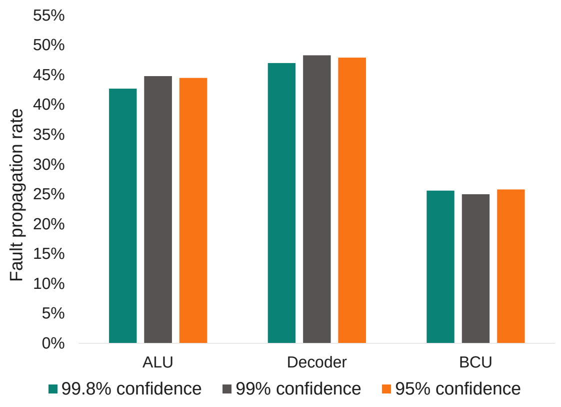
The findings from the charts in Figures 4-7 reveal that despite the variability in the confidence level of SFI campaigns for each component, the Fault Propagation Rates consistently fall within the acceptable error margin of 5%. This solidifies the effectiveness of the method employed. Notably, there is a slightly higher Fault Propagation Rate (ranging from 5-10%) observed in the 2-stage pipelined CPU in comparison to other benchmarks. This discrepancy can be attributed to the design's smaller area, which does not incorporate extra logic to prevent numerous faults as the 5-stage pipeline CPU does. Among all the components, the 2-stage pipelined CPU running the Dhrystone benchmark exhibits the highest failure rate, while the 5-stage pipelined CPU running the MD5 benchmark demonstrates the lowest failure rate. The decoder component emerges as the most vulnerable to failures, necessitating protective measures. Conversely, the branch control unit proves to be the least susceptible to faults due to its reliance on specific instructions like branch and jump types to activate faults. Nonetheless, it is evident from the results that the Fault Propagation Rate is influenced by the workload being executed.



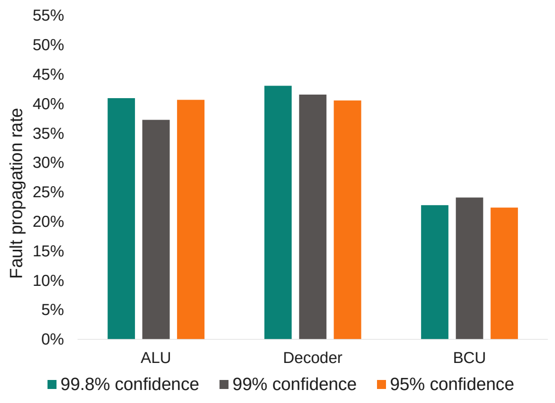
(a) Fault injection on 2-stage pipelined CPU

(b) Fault injection on 5-stage pipelined CPU

Fig. 4: Fault Propagation Rates using Dhrystone benchmark as firmware

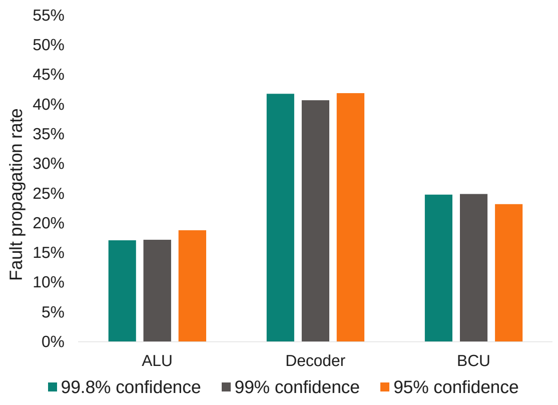


(a) Fault injection on 2-stage pipelined CPU

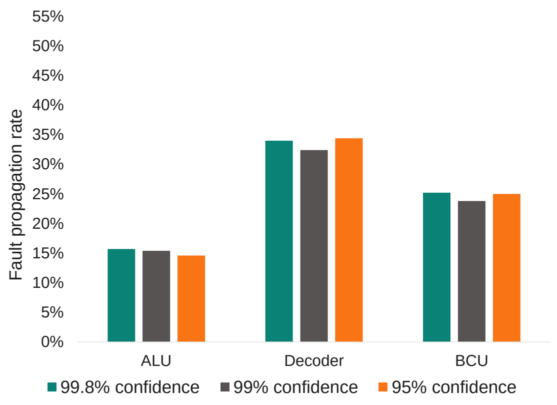


(b) Fault injection on 5-stage pipelined CPU

Fig. 5: Fault Propagation Rates using SHA-256 benchmark as firmware

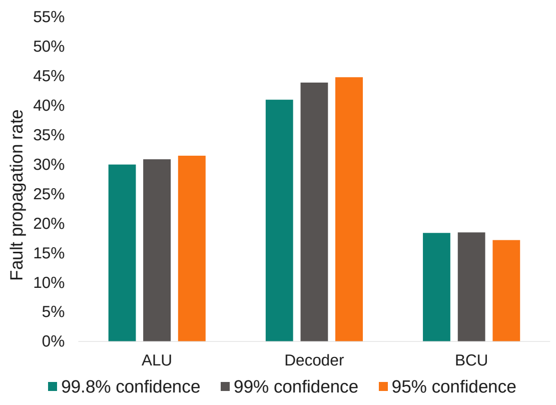


(a) Fault injection on 2-stage pipelined CPU

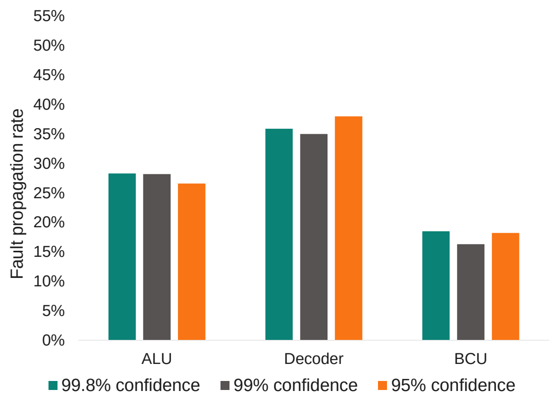


(b) Fault injection on 5-stage pipelined CPU

Fig. 6: Fault Propagation Rates using MD5 benchmark as firmware



(a) Fault injection on 2-stage pipelined CPU



(b) Fault injection on 5-stage pipelined CPU

Fig. 7: Fault Propagation Rates using CRC-32 benchmark as firmware

VII. Conclusion

This paper introduces an advanced approach combining statistical analysis and model-driven techniques to evaluate the Fault Propagation Analysis of different versions of RISC-V processors. To conduct comprehensive experiments, two separate RISC-V CPU subsystems were employed, and each was subjected to multiple trials utilizing four diverse workloads or benchmarks. The outcomes of these experiments demonstrated that the SFI technique proved to be remarkably successful, as the rates of fault propagation consistently remained within a margin of error range of 5% regardless of the varying confidence rates applied.

This study sets the groundwork for future investigations, which will involve conducting comparisons and assessments with other fault simulation tools, including both commercial and open-source options.

VIII. Acknowledgements

Part of the work has been performed in the project ArchitectECA2030 under grant agreement No 877539. The project is co-funded by grants from Germany, Netherlands, Czech Republic, Austria, Norway, France and Electronic Component Systems for European Leadership Joint Undertaking (ECSEL JU). Part of the work described herein is also funded by the German Federal Ministry of Education and Research (BMBF) as part of the research project Scale4Edge (16ME0122K).

References

[1] "Part 7: The 2022 wilson research group functional verification study," https://blogs.sw.siemens.com/verificationhorizons/2022/11/28/part-7-the-2022-wilson-research-group-functional-verification-study/, accessed: 2023-09-13.

[2] J. Schreiner, R. Findenig, and W. Ecker, "Design Centric Modeling of Digital Hardware," in *IEEE International High Level Design Validation and Test Workshop, HLDVT 2016*, 2016, pp. 46–52.

[3] W. Ecker and J. Schreiner, "Introducing model-of-things (mot) and model-of-design (mod) for simpler and more efficient hardware generators," in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Sept 2016, pp. 1–6.

[4] 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.

[5] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 172–182.

[6] J.-C. Baraza, J. Gracia, S. Blanc, D. Gil, and P.-J. Gil, "Enhancement of fault injection techniques based on the modification of vhdl code," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 6, pp. 693–706, 2008.

[7] J. Baraza, J. Gracia, D. Gil, and P. Gil, "Improvement of fault injection techniques based on vhdl code modification," in *Tenth IEEE International High-Level Design Validation and Test Workshop, 2005.*, 2005, pp. 19–26.

[8] D. Kammler, J. Guan, G. Ascheid, R. Leupers, and H. Meyr, "A fast and flexible platform for fault injection and evaluation in verilog-based simulations," in *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2009, pp. 309–314.

[9] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 172–182.

[10] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 622–629.

[11] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.

[12] Verilator, "Verilator," https://www.veripool.org/verilator/.