# Automated Thread Evaluation of Various RISC-V Alternatives using Random Instruction Generators

Endri Kaja*†, Nicolas Gerlin*†, Dominik Stoffel †, Wolfgang Kunz †, Wolfgang Ecker*‡
*Infineon Technologies AG
‡ Technische Universität München, Germany
*Email: Firstname.Lastname@infineon.com*
†Technische Universität Kaiserslautern, Germany
*Email: stoffel@eit.uni-kl.de, kunz@eit.uni-kl.de*

**Abstract**

**Safety-critical designs require fast, automated, and low-overhead verification techniques to cope with the ever-increasing complexity and shortening Time-to-Market constraints. Consequently, engineers are constantly looking for highly efficient verification approaches. In this paper, we propose an automated safety verification technique that combines Model-driven Fault Simulation and Model-driven Random Instruction Generation. The technique performs Exhaustive Fault Injection and Statistical Fault Injection with minimal effort, and at the same time provides acceptable fault coverage results with low-performance overhead. The proposed approach scales to a wide variety of RISC-V CPU subsystems and experimental results achieve a fault coverage rate of 40% up to 99% while conducting fault injection on different components of the CPU.**

## I. Introduction and background

The complexity and the size of the Systems-on-Chip (SoCs) are constantly increasing and, consequently, they are facing a higher risk of failure. Radiation, aging, mechanical stress, and processing defects could lead to undesired failure events. Safety-critical designs need to guarantee continuous, reliable, and safe operations even under harsh operating conditions, thus various fault detection and corrections mechanisms are required to mitigate the effects of hardware faults. The automotive industry has adopted ISO 26262 to standardize the implementation, verification, and validation of safety-critical designs.

*Fault injection* is the recommended technique to evaluate the dependability of the designs and it is defined as the observation of the behavior of the system in the presence of intentionally injected faults [1]. State-of-the-art fault injection techniques are divided into the following categories [2]: Hardware-based, Software-based, Simulation-based, Emulation-based, and Hybrid-based.

Improvement in fault injection processes is required to fulfill the ever-shortening Time-to-Market deadlines in the industry. Therefore, it is required to have a higher degree of automation to increase productivity. Ecker et al. [3], [4] presented the automated RTL generation framework, namely MetaRTL. MetaRTL is a proprietary framework that describes the components of RTL, e.g. logic gates, registers etc, and utilizes in-built python APIs to generate the RTL from top-level metamodel specifications. In addition, MetaRTL utilizes a library of components, which are classical building blocks for the generated designs. The framework reduces significantly the manual efforts and at the same time, increases design productivity. In this paper, an automated simulator-independent mixed granularity fault injector (gate level/register-transfer level) is utilized to inject different types of hardware faults [5]. The fault injection framework utilizes the above-mentioned RTL generation framework. Fig. 1 presents the generation flow of the design with fault injection capabilities using saboteur-based techniques.

Selected parts of the design that are subject to fault injection are kept on a gate-level (GL) granularity while the rest of the design is represented on a traditional register-transfer level (RTL) granularity. To achieve that, the Python script *ToD Generator* (Fig. 1) reads the data from the gate-level netlist and technology library and generates a new Template-of-Design (ToD) of mixed RTL/GL granularity. ToD is the microarchitecture blueprint of the intended design. As the next step, saboteurs are added to the Model-of-Design, which is a platform and technology-independent hardware description. Saboteur control lines are propagated as Primary Inputs (PI) and the existing RTL generation flow is used to generate the design in mixed granularity flavor. In the end, a formal equivalence check is applied to check whether the transformation process has introduced any bug in the new design. The presented flow enables fast RTL fault simulation without losing the accuracy of the gate-level fault simulation.

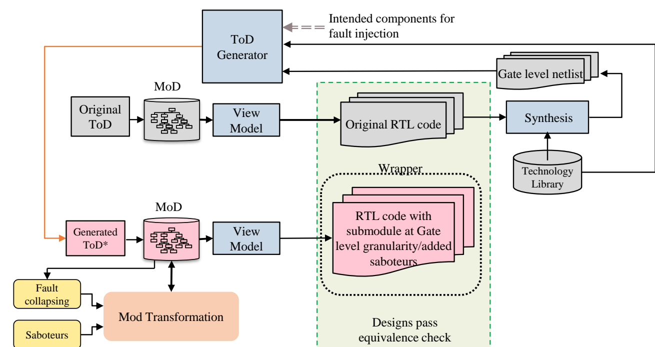



Fig. 1: Mixed register-transfer/gate level fault injector

Although fault simulation is advantageous in several aspects, an automated approach covering all aspects of fault simulation is required to keep the manual effort low. Safety-critical designs demand rigorous, stringent, and automated verification techniques with minimal effort. Moreover, complex testing techniques are required to achieve acceptable *fault coverage* according to different standards. Fault coverage is defined as the ratio of the detectable faults to the total faults and various approaches have been developed to increase the overall coverage. Nevertheless, the widely used techniques (e.g. Bult-in Self Test) suffer from performance and area constraints. To combat the existing drawback, we present in this paper an approach to combine model-driven hardware generation and model-driven fault simulation to automatically evaluate RISC-V designs' fault coverage with minimal manual effort.

## 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 [6] enables a formal fault injection and propagation analysis to support safety and security verification. Similarly, Fault Injection App [7] and Fault Propagation Analysis [8] from Onespin®provide formal verification methods to verify safety mechanisms. The interface enables the user to define custom fault scenarios, or pick predefined ones. It is necessary to mention that the previously described tools are vendor-specific. GenFi [9] is an approach from academia to inject faults on microarchitectural simulators such as MARSS and Gem5, while Baraza et al. [10] [11] modify the hardware description of the design utilizing saboteurs and mutants. In [12] authors propose a Verilog-based framework to perform fault injection based on Verilog Programming Interface (VPI).

## III. RISC-V CPU generation

The in-house model-driven RTL generation framework, MetaRTL, enables productivity increase thanks to the flexibility of the Python-based APIs. The approach provides freedom for various target HDLs and at the same time generates various design alternatives while utilizing the same Python code. The key element of the generation framework is the *metamodel*. A metamodel represents the structure of a model, its attributes, and the relation between them, i.e. it formalizes informal specifications to well-structured formal specifications. Fig. 2 represents the MetaRISC metamodel as a unified markup language (UML) class diagram. This metamodel is used to generate various RISC-V alternatives [13] depending on model configurations.

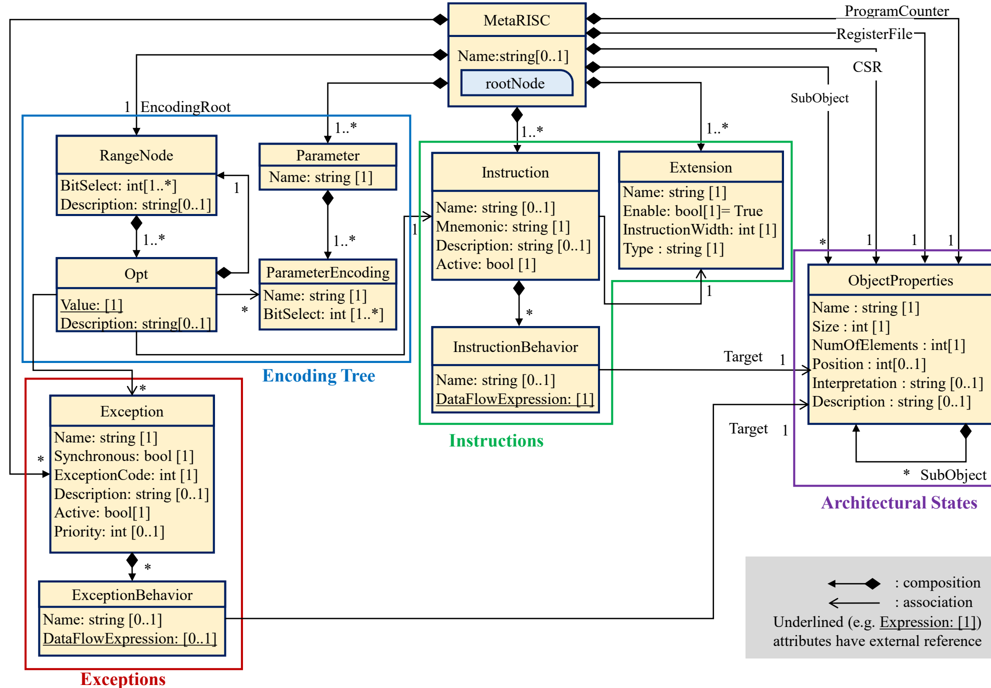



Fig. 2: RISC-V metamodel [13]

Four main components compose the MetaRISC: *Encoding Tree*, *Instructions*, *Architectural States*, and *Exceptions*:

- *Encoding Tree* builds a tree-like structure to determine the parameters and the format of all the instructions. The high degree of flexibility facilitates supporting different types of instruction sets.
- *Instructions* describe the instruction and its behavior, i.e. the changes that the instruction prompts in the architectural state of the CPU.

- Architectural States represent the important state elements of the CPU such as Program Counter (PC), General Purpose Registers (GPR), and Control and Status Registers (CSR).
- *Exceptions* define the set of exceptions (unexpected events) and their behavior.

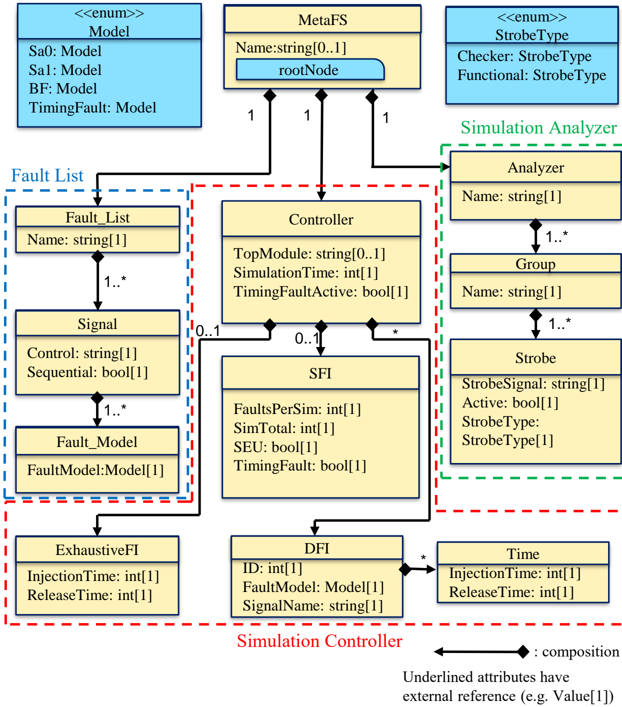
The user needs to fill the model with the desired configurations, and by using MetaRTL, various RISC-V alternatives can be generated.

## IV. Model-based fault simulation framework

The flexibility of the generation framework facilitated the development of the automated and configurable Model-driven Fault Simulation Framework [14]. The framework extends and builds upon the fault injection flow presented in Section I. Fault simulation framework, MetaFS, is presented in Fig. 3a. MetaFS is composed of three main components: *Simulation Controller*, *Fault List*, and *Fault Analyzer*.

*Simulation Controller* allows the user to define total simulation time, enable timing faults, and choose different fault injection campaigns such as Statistical Fault Injection (SFI), Exhaustive Fault Injection (EFI), and Direct Fault Injection (DFI):

- During the SFI campaign, only a certain random subset of all possible errors is injected and the design's safety dependability is measured with a certain degree of confidence. SFI class permits the user to determine the number of faults to inject per simulation and the total amount of simulations. Further, boolean attributes of this class activate SEU (Single Event Upset) and Timing faults. All faults are injected randomly complying with user configurations.
- During the EFI campaign, all stuck-at faults are injected into all design signals. The user is able to choose injection and release time using the EFI class.
- Attributes of the DFI class determine injecting predefined fault models into predefined signals during certain clock cycles.




(a) Fault simulation metamodel

```
1  always @(i, k) begin
2      if (!fault_injected) begin
3          if (i==1) begin
4              if (k >= 50 && k <= 52) begin
5                  if (!wr_flag) begin
6                      $fwrite(fault_sim_report, "%s%s", unify_char_nr("1", "post", 13),
7                      unify_char_nr("50", "post", 21));
8                      wr_flag = 1;
9                  end
10                 FI_Control_All_834 = 3'b101;
11                 bf_end = 1;
12             end
13             else if (bf_end) begin
14                 FI_Control_All_1 = 3'b100;
15                 fault_injected = 1;
16                 bf_end = 0;
17             end
18         end
19          else if (i==2) begin
20             if (k >= 121 && k <= 650) begin
21                 if (!wr_flag) begin
22                     $fwrite(fault_sim_report, "%s%s", unify_char_nr("11", "post", 13),
23                     unify_char_nr("121", "post", 21));
24                     wr_flag = 1;
25                 end
26                 FI_Control_All_828 = 3'b000;
27                 fault_injected = 1;
28             end
29         end
30         else if (i==3) begin
31             if (k >= 640 && k <= 642) begin
32                 if (!wr_flag) begin
33                     $fwrite(fault_sim_report, "%s%s", unify_char_nr("3", "post", 13),
34                     unify_char_nr("640", "post", 21));
35                     wr_flag = 1;
36                 end
37                 FI_Control_seq_50 = 3'b101;
38                 bf_end = 1;
39             end
40             else if (bf_end) begin
41                 FI_Control_seq_50 = 3'b100;
42                 fault_injected = 1;
43                 bf_end = 0;
44             end
45         end
```

(b) Generated testbench

Fig. 3: Automated testbench generation

A Python script collects the data from the design and provides the necessary information to fill the attributes of the *Fault List* class. The *Signal* class defines design signals and the respective saboteur control line to inject the fault. *Fault_Model* determines all possible fault models to inject due to the fact that fault collapsing eliminates redundant faults.

*Simulation Analyzer* provides the necessary information to analyze and classify the injected faults. The user fills metamodel attributes with desired design signals (internal signals/registers as well as primary outputs) to analyze via the *Strobe* class. Additionally, it is possible to specify whether the selected signal is a functional output or an output of a safety mechanism.

The framework generates SystemVerilog and C++ testbench in accordance with the metamodel data. Fig. 3b represents a snippet of a generated fault simulation testbench. Lines 3, 19, and 30 represent the simulation count, and in this case, the user

has selected to inject only three faults. Lines 4, 20, and 31 define the timepoints when the fault should be injected and lines 10, 26, and 37 determine the fault model to inject, i.e. random saboteur control lines are driven to inject a particular fault model. During simulation 1 and 3 a bitflip is injected since the fault is kept high only for one clock cycle, and during simulation 2 the fault is kept high for the whole duration of the simulation. The rest of the lines store the simulation data in a log file.

Fig. 4 describes the complete fault simulation flow on a block level. After generating the DUT utilizing MetaRTL, the fault simulation framework generates the necessary testbench to a perform fault simulation campaign. After storing the data in the log file for each faulty simulation, the testbench compares this data to the golden non-faulty simulation. This comparison classifies the faults according to their effect on the design behavior. As the last step, the fault classification results are stored in a log file. A snippet of this log file is shown in Fig. 5 and each column represents the following data:

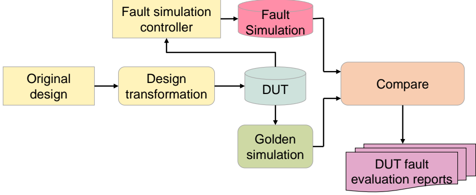



Fig. 4: Fault simulation flow

- Fault_Sim: fault simulation count.
- Fault_Inject_Time: fault injection time.
- Fault_Propagate_Time: it shows the timepoint when the fault has propagated to any of the strobes selected by the user. If the fault did not propagate, the symbol *na* will be displayed.
- Propagated_FStrobe: displays the strobe signal where the fault propagated.
- Fault_Detect_Time: displays the timepoint when the fault was detected by a safety mechanism. If not, *na* will be displayed.
- Detected_CStrobe: displays which checker strobe detected the injected fault.

```
Fault_Sim  Fault_Inject_Time    Fault_Propagate_Time     Propagated_FStrobe                                   Fault_Detect_Time    Detected_CStrobe


1          50                   101                      comp_TopSoC.comp_MinimumSoc.comp_CoreGen.IB_addr_out     na                   Fault not detected by checker strobe
2          50                   177                      comp_TopSoC.comp_MinimumSoc.comp_CoreGen.DB_wdata_out    na                   Fault not detected by checker strobe
3          50                   na                       Fault did not propagate                                  na                   Fault not detected by checker strobe
4          50                   101                      comp_TopSoC.comp_MinimumSoc.comp_CoreGen.IB_addr_out     na                   Fault not detected by checker strobe
5          50                   101                      comp_TopSoC.comp_MinimumSoc.comp_CoreGen.IB_addr_out     na                   Fault not detected by checker strobe
Safe Undetected Faults: 1
Safe Detected Faults: 0
Dangerous Undetected Faults: 4
Dangerous Detected Faults: 0
Total Faults: 5
```

Fig. 5: Fault simulation results

At the end of the file, the collection of this data will be shown:

- Safe Undetected Faults: faults that did not propagate at any strobe.
- Safe Detected Faults: faults that are detected and fixed by the safety mechanism.
- Dangerous Undetected Faults: faults that propagate to the functional strobes but are not detected by the safety mechanism.
- Dangerous Detected Faults: faults that propagate to functional strobes but are detected by the safety mechanism.

It is necessary to note that this approach is design-independent, i.e. fault injection can be applied to all kind of designs generated by MetaRTL (e.g. peripherals, safety mechanisms, etc.).

## V. Random instruction generation

Safety-critical designs require rigorous verification and testing to comply with different international standards. Automatic Test Pattern Generation (ATPG) and Built-in Self-test (BIST) are the most common techniques that provide adequate input test sequences for the dependability evaluation of safety-critical designs. Even though the techniques provide very high fault coverage results, they suffer from area and performance constraints. ISO 26262, the international standard for automotive safety, recommends various Automotive Safety Integrity Levels (ASILs) for different designs. Consequently, a low-overhead random input test sequences can be utilized to achieve acceptable fault coverage results. The *Model-driven Fault Simulation Framework* is combined with *Model-driven Random RISC-V Instruction Generation* as input test stimuli to automatically evaluate fault coverage of various RISC-V-based CPU subsystems.

The key element of the random instruction generator is the metamodel presented in Fig. 2a. A Python script reads the model data and generates the instructions accordingly, i.e. generates the valid set of instructions as defined in the model from Section III. The generator script is highly configurable, and thus, the user can constrain the test. In the following, the configurable parameters are shown:

- Constraints: the user can constrain the class of the instructions to generate, e.g. only "load" and "csr" type instructions. Further, it is possible to constrain instructions to have read-after-write dependencies.
- Length: the user can specify the number of instructions to be generated. The varying length of the test sequence is an important factor that impacts the fault coverage results.

- File Type: supports two different file types such as hex and bin.
- Memory Start Address: the user can specify as an integer the starting address of the data memory.

After configuring the generator, the framework APIs are utilized to read model data and to generate a set of random instructions which serve as test input for the RISC-V CPU.

## VI. Application and Results

The applicability and the low cost of combining random instruction generation with fault simulation flow have been demonstrated on a CPU subsystem with various RISC-V CPU alternatives. The subsystem is composed of the CPU, program, and data memory instances, peripherals such as SPI, UART, Interrupt Controller, and a bus fabric with bus bridges. The tests are run on three similar subsystems with different RISC-V alternatives, i.e. only the CPU changes. The input test (see Section V) is loaded into the instruction memory and varying lengths of test sequence have been applied. The following subsections present the fault coverage results of the CPU while applying random instructions as test stimuli on three RISC-V CPU subsystems, respectively RV32IMC, RV32IMC-Exceptions, and RV32IMC-MAC. As test input length of 100, 150, 200, 250, and 300 instructions was utilized. Faults were injected only on particular components of the CPU such as the Arithmetic-Logic Unit (ALU), Forwarding Unit (Fw Unit), Prefetcher, and Fetch stage. Fig. 6-8 display fault coverage results of the EFI campaign and the total amount of injected faults is shown on the right side of the figures. Fig. 9 presents the SFI campaign results. All the campaigns were run on Xcelium®simulator.

### A. RV32IMC EFI campaign

As shown in Fig. 6 the fault coverage increased constantly with a higher set of inputs for all components faults except for the ALU faults. Further, Forwarding Unit faults cause up to a 99% fault coverage rate for an input set of 300 random instructions. The steepest incline of fault coverage was observed while injecting faults at Prefetcher, but it stabilizes after 250 instructions. Interestingly, the input stimuli do not sensitize many fetch stage faults, and one reason for that is random jumps and branches to out-of-bound memory locations.

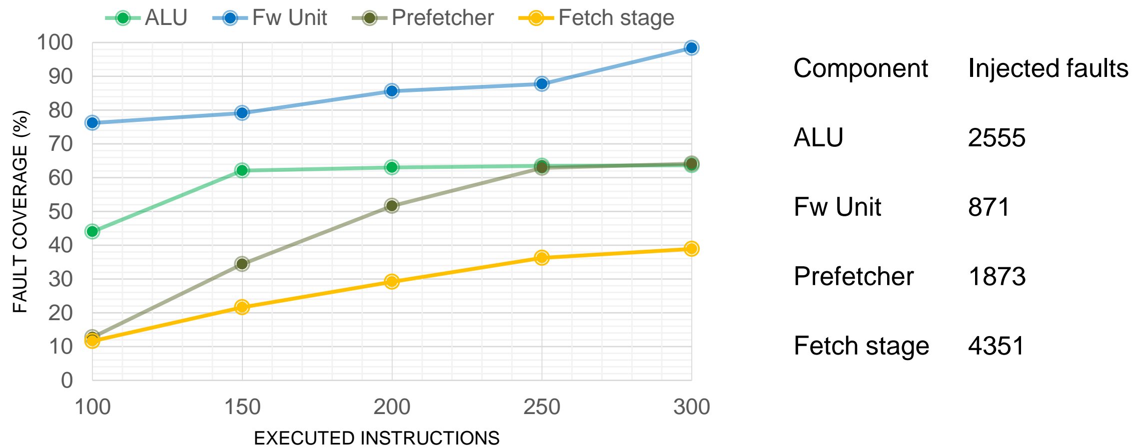



Fig. 6: RV32IMC EFI campaign

### B. RV32IMC-Exceptions EFI campaign

An Exception Unit (EU) is added to the CPU during this campaign to support exception handling. Similar results to the RV32IMC EFI campaign were observed as presented in Fig. 7. A slight increase of fault coverage was noticed on all components except the Forwarding Unit where it remained the same. Further, the total amount of injected faults of the Fetch stage is higher due to the added EU.

### C. RV32IMC-MAC EFI campaign

The EFI campaign is applied to the CPU with support for exceptions and MAC extension. In Fig. 8, there are noticed almost identical results to RV32IMC-Exceptions EFI campaign (see Fig. 7) with a slight increase of fault coverage due to ALU faults.

### D. SFI campaign

During the SFI campaing, there were injected random fault models, including stuck-at, SEU, and Timing faults, on random fault locations during random timepoints. The test length was 300 instructions and Fig. 9 presents the fault simulation results on all components like in previous campaigns. Fault coverage has significantly reduced compared to EFI campaign, thus, SEU and Timing faults have had a low impact on the selected components.

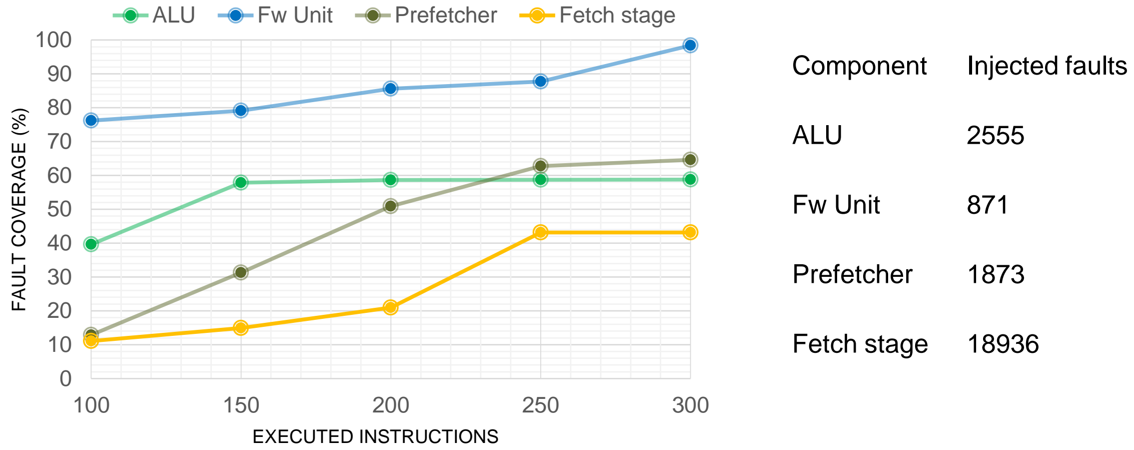



Fig. 7: RV32IMC-Exceptions EFI campaign

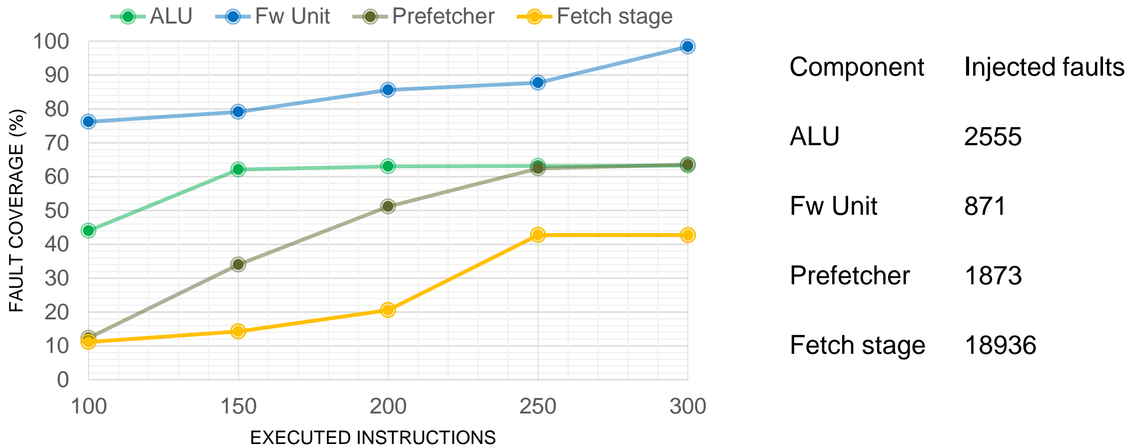



Fig. 8: RV32IMC-MAC EFI campaign

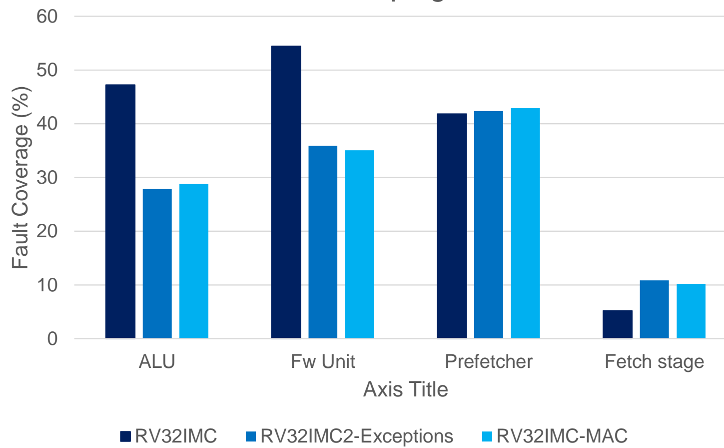



Fig. 9: SFI campaign of three CPU variants

## VII. Conclusion and Future Work

In this paper, we introduced a low-overhead automated safety evaluation of various RISC-V alternatives by combining Model-driven Fault Simulation and Model-driven Random Instruction Generation. The framework is generic and fault injection can be applied to various distinct designs. Numerous experiments were conducted on three different RISC-V CPU subsystems with varying lengths of the test sequences. The Exhaustive Fault Injection campaign resulted in an acceptable fault coverage range (40%-99%) for a length of input test of 300 instructions, i.e. with a minimal memory footprint. The total effort required to run all the campaigns was 1 person-day. Extending the supported fault models, constraining jumps to in-bound memory

locations, and further comparisons to other commercial and open-source fault simulation tools are subject to future work. Due to internal policy, the framework is not intended to be provided as open source in the near future.

## 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] 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] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, July 2004.

[3] 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.

[4] 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.

[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] "FSV," https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-functional-safety-verification-app/, [Online: Accessed on 10.October.2022].

[7] "Fault Injection App," https://www.onespin.com/products/specialized-apps/fault-injection-app/, [Online: Accessed on 10.October.2022].

[8] "Fault Propagation Analysis," https://www.onespin.com/products/specialized-apps/fault-propagation-analysis/, [Online: Accessed on 10.October.2022].

[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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] E. K. et al., "Metfi: Model-driven fault simulation framework," https://arxiv.org/abs/2204.13183, 2022.