# MSc Thesis

FPGA-based Fault Emulation for Safety Critical ICs

Nivedita Lohar

TUDelft

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

# MSc Thesis

## FPGA-based Fault Emulation for Safety Critical ICs

by

Nivedita Lohar

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday, September 22, 2023, at 14:00 hours.

| | | |
|---|---|---|
| Student Number: | 5483565 | |
| Project Duration: | November 2022 - September 2023 | |
| Thesis committee: | Prof. Dr. Ir. Said Hamdioui | TU Delft, supervisor |
| | Dr. Ir. Moritz Fieback | TU Delft, supervisor |
| | Dr. Ir. Caspar Van Vroonhoven | Analog Devices, supervisor |
| | Alessandro Trevisan | Analog Devices, supervisor |
| | Dr. Ir. Rene van Leuken | TU Delft, committee member |

TU Delft

# Abstract

In 2022, there were over 26 million electric automobiles on the road, a 60% increase with regard to 2021 and more than 5 times the stock in 2018. As automobiles become more electric and systems get increasingly complex, the safety requirements get more stringent. In 2011, the International Organization for Standardization (ISO) established ISO26262 to provide guidance to the semiconductor industry on the development process of safety essential ICs for automotive applications.

When evaluating the safety of a system parallel to the normal functional operation, traditional Design-for-Test (DFT) techniques such as scan chain, Built-In Self-Test (BIST), Joint Test Action Group (JTAG), and boundary scan are no longer viable options for two fundamental reasons. Firstly, safety assessment in the context of automotive applications necessitates evaluation at an application level, going beyond the capabilities of these techniques, which are primarily designed for structural and functional testing. Secondly, during safety checks, it is imperative that the normal operation of the integrated circuit (IC) remains uninterrupted, as the chip is often deployed in critical, real-time systems. These traditional DFT methods, while effective during the IC manufacture and production before they are released in the market, fall short of addressing the dynamic and application-specific safety concerns that arise during the operational lifecycle of safety-critical systems in sectors like automotive engineering.

To address these challenges, fault injection has emerged as the necessary step for safety assessment. ISO26262 explicitly recognizes fault injection as one of the most popular techniques for evaluating a system's safety and determining its Automotive Safety Integrity Level (ASIL). The safety metrics are required for certification of a product with ASIL. Fault injection allows for the creation of realistic fault scenarios and the assessment of how the system responds to these faults during normal operation, aligning more closely with the dynamic and application-specific nature of safety-critical systems in fields of automotive engineering.

Currently, many EDA companies provide Failure Mode Effects and Diagnostic Analysis (FMEDA) platforms for IC safety evaluations. However, these tools are time-consuming and resource-intensive. Additionally, as IC designs become more intricate, there is a reliance on fault reduction techniques such as statistical sampling, which entails simulating only a mere 5% of the overall fault space. Thus, an FPGA-based fault emulation system emerges as a promising approach to expedite this process.

The novelty of this work was in designing a dedicated platform tailored for the evaluation of safety-critical systems for automotive applications such as Battery Management Systems (BMS). This platform can execute the safety sequences on the system to evaluate the safety mechanisms implemented in the design in the presence of random faults assuming the chip is in use in the car. Moreover, the fault emulation activity provides the evidence necessary for the certification of products with ASIL level. Furthermore, performing this activity during the development stage helps in designing the ICs with the highest level of safety. The proposed FPGA-based fault emulation system efficiently overcomes three key challenges: it decreases execution time dramatically provides a speed-up of 296x compared to the simulation method, optimizes resource utilization, eliminates the tool license cost, and removes the requirement for considerable fault space reduction. This platform can emulate a large fault population of up to one million faults in less than three hours.

# Acknowledgements

As I write this section, marking the final step of my journey at TU Delft, I feel nostalgic, recalling the very first day I arrived in the Netherlands. These two years have been filled with a wealth of learning and emotions, and I would like to take this opportunity to express my profound gratitude to all the individuals who have made this journey a memory to cherish for a lifetime.

First and foremost, I extend my heartfelt thanks to my supervisors at TU Delft, Prof. Dr. Ir. Said Hamdioui and Dr. Ir. Moritz Fieback. They not only assisted me in resolving my various doubts related to the subject matter during my thesis but also groomed me to become a better engineer, well-prepared to tackle real-world challenges beyond the university. Their constant encouragement to think critically has equipped me to explore beyond the confines of this work.

I would also like to express my deepest appreciation to my supervisors at Analog Devices, Dr. Ir. Caspar Van Vroonhoven and Alessandro Trevisan. Their curiosity and active involvement in my work not only facilitated the timely completion of my thesis but also provided me with insights that extend far beyond the academic realm, into the industrial domain. This experience has endowed me with a unique perspective, allowing me to approach problems from both academic and industrial viewpoints. Additionally, I am grateful to all my colleagues at Analog Devices, who not only welcomed me into their professional family but also generously shared their valuable experiences, contributing significantly to my learning curve.

I am thankful to all my peers and friends who have been a part of this remarkable journey. Their presence turned this academic adventure into a joyful and enriching process, without which it would have been an arduous path to traverse.

Lastly, I wish to express my heartfelt gratitude to my parents and all my friends back in my home country for their unwavering support and constant availability throughout this journey.

This Acknowledgment section of the master's thesis captures the essence of the beautiful journey it has been.

*Nivedita Lohar*
*Delft, September 2023*

# Contents

# List of Figures

# List of Tables

# 1 Introduction

The Electric Vehicle was invented in 1834. However, due to the limitations associated with the batteries and the rapid advancement in Internal Combustion Engines vehicles, EVs almost vanished from the scene until the early 1970s. In the late 20th century due to the energy crisis, the interest in EVs was rekindling [10]. In the last 2 decades, the electric vehicle markets have expanded at an exponential rate, with sales exceeding 10 million in 2022. Electric vehicle sales have more than tripled in three years, from roughly 4% in 2020 to 14% in 2022. Figure 1.1 shows the trends in sales of Electric cars from 2016 to 2023. EV sales are predicted to remain sturdy through 2023. Over 2.3 million electric vehicles were sold in the first quarter, representing a 25% increase over the same period the previous year. Sales are projected to reach 14 million by the end of 2023, indicating a 35% growth year on year. As a consequence, electric vehicles account for 18% of overall vehicle sales [20].

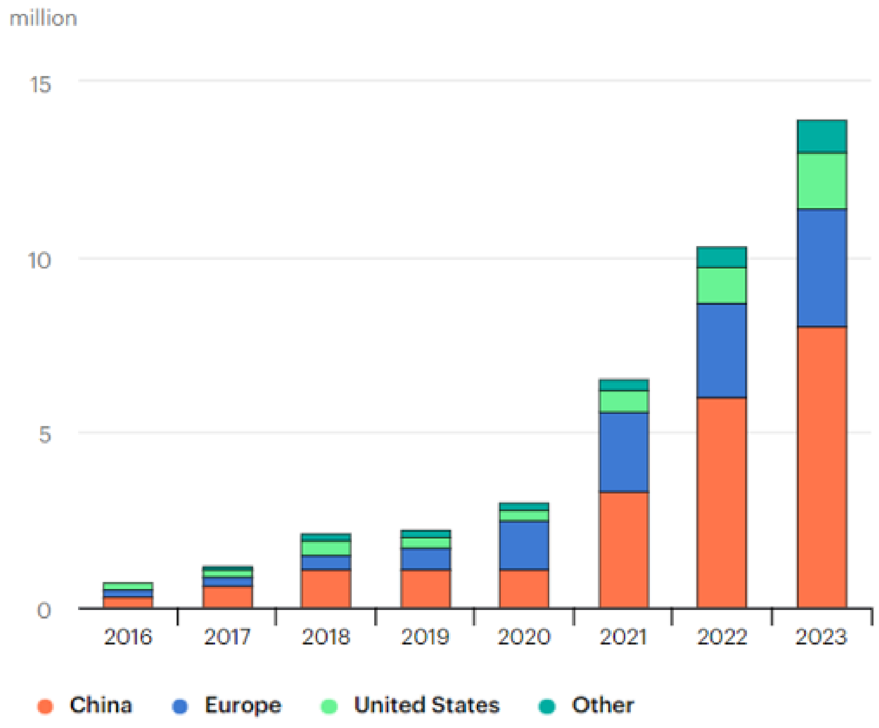


**Figure 1.1:** Trends in sales of Electric cars, 2016-2023[20]

The electronics in these EVs must meet high functional safety and reliability requirements. Traditionally, automotive manufacturers and system suppliers were in charge of functional safety criteria. However, as the complexity of electronics grows, the responsibility for addressing functional safety is also spreading along the supply chain to semiconductor businesses and design tool suppliers. Functional Safety (FuSa) is the idea of ensuring that automobile functions work reliably, even when something unexpected or dangerous occurs, thus protecting individuals from unacceptable risk or damage.

In 2011, the International Organization for Standardization developed ISO26262, which provides guidelines on safety-related electronic and electrical systems in automobiles. This document [51] defines a set of regulations that automotive chip development companies must follow in order to certify their products for use in commercial vehicles. Furthermore, ISO26262 defines a fault classification system known as Automotive Safety Integrity Levels (ASIL) with the goal of reducing potential dangers caused by the faulty behavior of the electronics [50]. There are four levels of ASIL A, B, C, and D with D being the highest degree of safety level.

In order to certify the products with ASIL level, there is evidence required that the system developed for safety-related applications complies with all ISO26262 requirements. Currently, ISO26262 considers fault injection as one of the mandatory steps in system validation activities. The industry now uses a range of safety tools from EDA companies for Failure Modes, Effects, and Diagnostic Analysis (FMEDA). However, as designs get more sophisticated, the fault injection process using simulations might take days or even months.

The primary goal of this project is to accelerate fault injection by developing an FPGA-based emulation-based platform for testing system safety in the presence of random faults. The rationale for the need for such a platform is discussed in this chapter. Furthermore, various different techniques of testing a system are explored, and it is emphasized that fault injection is necessary for assessing a system, particularly for safety purposes. Following that, the significance of Functional Safety and the various processes used in the development of safety-critical ICs are demonstrated. Second, the most recent strategies in the field of fault injection are discussed. Finally, the project's purpose and obtained results are presented.

## 1.1. Motivation

Design for Testability (DFT) is a critical aspect of modern integrated circuit (IC) design, aiming to ensure that electronic components and systems are easy to test and diagnose for faults or defects [17]. DFT emerged in the mid-20th century as semiconductor technology advanced and integrated circuits became increasingly complex. Engineers and designers recognized the need for efficient testing methods to identify manufacturing defects and ensure the reliability of electronic systems. As ICs grew in complexity, traditional testing techniques proved inadequate, leading to the development of DFT [42].

In the early days of DFT, designers mainly relied on manual techniques to improve testability. These included adding test points to the circuit layout, simplifying circuit structures to enhance observability and controllability, and ensuring that faults were more easily detectable [63]. These rudimentary approaches laid the foundation for more sophisticated DFT methodologies.

One of the seminal developments in DFT came with the introduction of scan design in the 1970s [65]. This method involved incorporating shift registers into the circuit to control the state of all flip-flops, allowing for easy access and control during testing. Scan design significantly improved testability and became a cornerstone of DFT. In the 1980s, the concept of Built-In Self-Test (BIST) gained prominence [27]. BIST techniques involve embedding test generators and response analyzers within the IC itself, enabling autonomous testing without external equipment. BIST reduced the reliance on external testers and contributed to cost-effective testing. The Joint Test Action Group (JTAG) standard, introduced in the 1990s [64], provided a standardized approach to testing and debugging PCBs and ICs. JTAG utilizes a boundary scan chain to access and test various components on a board, enhancing testability for complex systems. As ICs continued to grow in complexity, hierarchical DFT techniques emerged. These approaches involved dividing the design into smaller manageable blocks, each with its own DFT structures. Automatic Test Pattern Generation (ATPG) tools were developed to generate test patterns for these hierarchical structures, reducing the complexity of test generation.

In the realm of safety-critical integrated circuits (ICs) especially for automotive applications, the challenge lies not only in ensuring their functionality but also in verifying their performance under the most demanding real-world conditions where random failures could occur. Random faults are the ones that can arise unexpectedly throughout a hardware element's lifespan. When ICs are embedded in systems that must operate seamlessly in situations such as self-driving cars navigating busy streets delivering life-saving treatments, conventional Design for Testability (DFT) techniques face limitations. These traditional methods, primarily tailored for controlled lab environments or manufacturing testing, often fall short in simulating the dynamic and unpredictable scenarios encountered in operational use. Consequently, testing safety-critical ICs while they perform their normal functions presents a unique conundrum. In such cases, the most effective approach to guaranteeing their reliability and safety is through fault injection testing conducted in parallel to their standard operation [55]. This methodology allows for the rigorous assessment of ICs responses to random faults, ensuring they meet the stringent safety requirements of ISO26262.

Moreover, in safety-critical applications like autonomous vehicles, ICs must perform their functions while the system is in motion. Testing these ICs using DFT methods typically requires halting or isolating them from the operational system, which doesn't provide a realistic assessment of their performance under actual conditions. The battery management system ICs used in hybrid vehicles, for example, continually measure the voltage and current of the battery cells. To test this system for random single-point faults, safety procedures are run in tandem with battery readings to guarantee the system is not experiencing overvoltage or overcurrent [47].

To assure the functional safety of the products, several techniques for design known as Safety Mechanisms (SM) such as logic redundancy, multiple processing paths, triple modular redundancy, fail-safe mechanisms, and fault-tolerant patterns are incorporated in the safety-critical ICs [19]. These safety features must be examined for random faults assuming the vehicle is in motion in order to provide evidence for ASIL certification. Furthermore, this serves as proof of safety products to the OEMs, Tier 1 and Tier 2 customers. OEM stands for Original Equipment Manufacturer; these are the ultimate product makers who deal with the vehicle's final market release. Tier 1 suppliers create solutions that are tailored to the completed product without substantial changes. Tier 1 must, however, employ parts that allow for component manufacture. Tier 2 joins the picture here. These customers utilize these safety-critical ICs.

To assess the safety mechanisms (SMs), various sets of safety sequences are created [43]. These sequences are detailed in a document referred to as the Safety Manual, which is provided to the customers for monitoring the ICs. During the verification process, fault injection is performed to evaluate the Failure Modes, Effects, and Diagnostic Analysis (FMEDA). During fault injection, random faults such as single-point, dual-point, or transient faults are injected into the system and various safety routines are executed to examine the behavior at an application level [49][61].

Currently, the industry employs a variety of safety tools from EDA firms, such as Cadence's Xcelium Fault Simulator, a simulation tool used for executing Digital Fault Injection (DFI). Synopsys also offers a VC Functional Safety Manager, which is a single platform for doing FMEDA. However, fault Injection using a simulation tool becomes extremely time-consuming as the circuit complexity increases and the fault space grows a few hundred thousand. These simulations take weeks or even months for a fault space of 1M faults. As a result, there is a need to limit the amount of faults for simulations without affecting the quality of the results. Statistical sampling [59], fault collapse [62], and formal approaches [6][5] are some of the techniques used to reduce fault space. Statistical sampling, for example, aids in minimizing the fault space by simulating only 5-10% of the total fault population. Though this method guarantees that the results are unaffected, it is assumed that these defects represent the whole fault space.

Furthermore, most modern systems are mixed signal in nature, and as complexity rises, simulations using Analog Mixed Signal (AMS) Fault Injection and Digital Mixed Signal (DMS) Fault Injection become exceedingly cumbersome when compared to fully digital fault injection. This is because of the inherent complexity of analog and mixed-signal circuits, the need for continuous-time modeling, and convergence challenges that contribute to slower simulation times compared to fully digital designs. This necessitates the development of an environment that allows for hardware acceleration.

There are two ways of achieving hardware acceleration, one hardware emulators provided by the EDA companies and secondly, the FPGAs. This emulation platform should be capable of modeling the

Design Under Test, performing fault injection on the modeled hardware, and evaluating the system by running the functional and safety . From the literature study, it was observed that for fully digital circuits, the execution time grows almost linearly with the increase in the fault space. Secondly, the DMS simulations are faster than the AMS simulations because, in DMS, the analog circuits are accurately modeled using the SystemVerilog user-defined datatype EEnet and Real Number Modeling (RNM) which provides acceleration over the actual schematics used in AMS simulations.

In the emulation world, particularly with FPGAs, significant hardware acceleration has been accomplished. However, there is some non-recurring cost at the start of the campaign owing to transferring the design from ASIC flow to FPGA. Including additional logic to physically incorporate defects into the design. Moreover, as seen in the graph, there is significant latency involved even for smaller fault spaces. This is due to the campaign configuration on the FPGA before executing the fault injection. However, as the fault space grows, the achieved speedup outweighs the simulation methods. As a result, this project leverages the use of hardware acceleration gained using FPGAs. It focuses on developing a platform capable of doing fault injection in order to assess the system's safety by running the safety sequences and evaluating the system in the presence of random faults. The primary objective is to reduce the execution time of fault campaigns. Furthermore, this activity aids in resolving the limited availability of resources and allows emulation of the complete fault population. For this experiment, the performance assessment of the execution time for 1M fault space was considered.

In the next parts, the requirement of automotive functional safety in IC design is introduced. Following this, some of the prior work in performing fault injection is discussed. Lastly, the scope of this project and the expected outcome are presented.

## 1.2. Automotive Functional Safety in IC design

Safety in technological development is not a novel notion. It was widely used in applications such as aerospace, military, and industrial machinery. As the technology progressed, the next-generation automobiles are increasingly becoming electrified, and the introduction of self-driving automobiles with Advanced Driver Assistance Systems (ADAS) drove the silicon designers towards creating even more sophisticated automotive integrated circuits. These chips must operate as intended, without any malfunctioning. This brings us to the realm of the Functional Safety (FuSa) concept. FuSa describes the detection of potentially hazardous situations and the activation of preventative procedures to mitigate them. For instance, if the car crashes in an accident, the sensors must detect the severity correctly and deploy the airbags within a few milliseconds. Even in case of malfunction, the system should return to a safe state where there is no unreasonable level of risk.

In 2011, the International Organization for Standardization (ISO) published ISO26262 [1], an international functional safety standard that protects the safety of a vehicle's electronics and electrical systems that are installed in series production passenger cars with a maximum gross vehicle mass up to 3500 kg. In 2018, the ISO26262 standard was amended to include all road vehicles, with new requirements reflecting current technological advancements.

ISO26262 specifies many procedures for determining how FuSa should be managed. One of these activities is determining the Automotive Safety Integrity Level (ASIL) that will be applied to the application. It assigns a grade from A to D, with D being the highest safety-critical level that must adhere to the most stringent testing. For example, Components like rear lights require merely an ASIL A rating. Whereas Headlights and brake lights are often ASIL B, whilst cruise control is typically ASIL C. And the airbags, anti-lock brakes, and power steering need an ASIL D grade, the highest level of rigor devoted to safety assurance because the hazards associated with their failure are the greatest.

One of the example of ASIL D design is the Battery Management System (BMS) used in electric cars to monitor cell voltage and current. This is illustrated below. Figure 1.2 presents a circuit diagram of a battery monitoring unit. In this figure, the BMS IC is the core of the monitoring circuit. The 8-bit AMTEL microcontroller employs a SPI bus to communicate with this IC and the various digital IOs. The microcontroller's job is to enable Controller Area Network (CAN) bus connection between the monitoring IC and the Pack Management Unit (PMU), regulate cell data acquisition, and monitor the balancing process. The microcontroller executes different functional and safety sequences in regular time intervals to monitor the State of Health (SOH) and State of Charge (SOC) for the safety of the

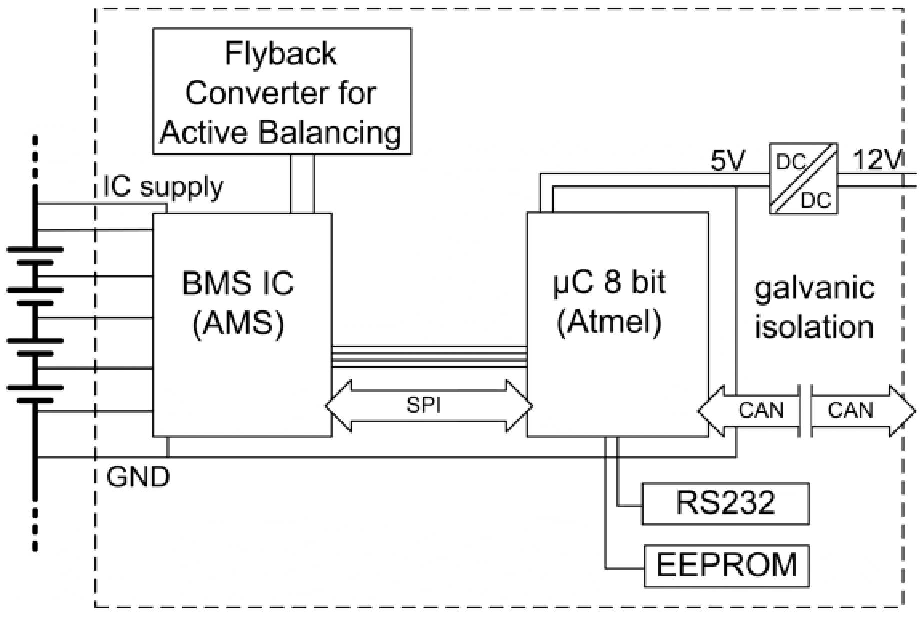


**Figure 1.2:** Battery Monitoring and balancing circuit overview [8]

batteries.

## 1.3. State of the Art

The fault injection technique goes back to the 1970s when it was initially employed to trigger hardware problems. Hardware Implemented Fault Injection (HWIFI) is a form of fault injection that seeks to replicate hardware breakdowns within a system. The initial hardware failure tests consisted of simply shorting connections on circuit boards and watching the effect on the system. Currently, there are four major categories of fault injection, namely, Hardware-based fault injection, Software-based fault injection, Simulation-based fault injection, and Emulation-based fault injection[29][69][36]. The overview of fault injection techniques is shown in Figure. 1.3.

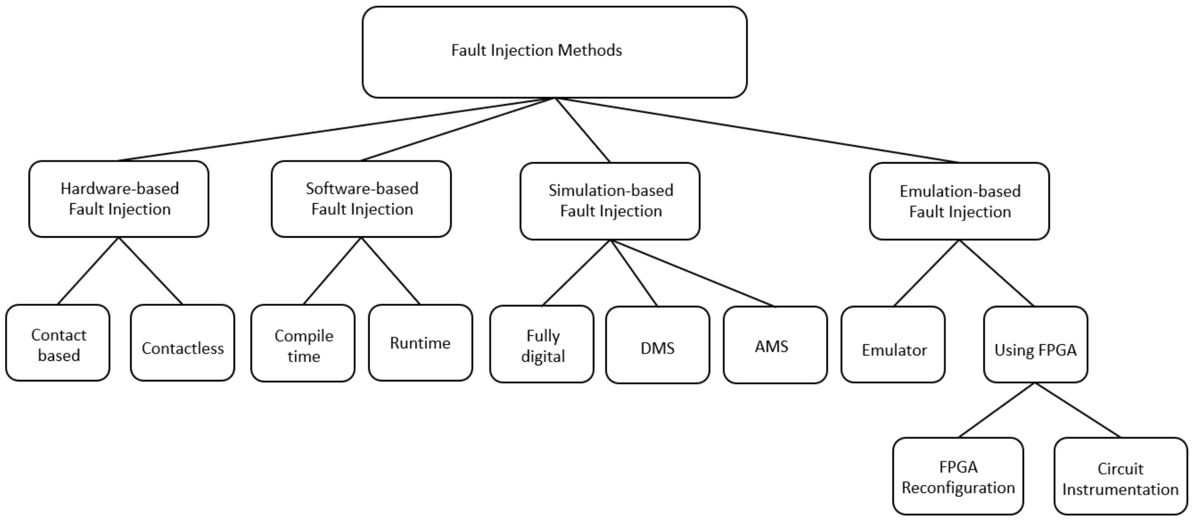


**Figure 1.3:** Overview of Fault Injection Techniques

The purpose of **Hardware-based fault injection** is to generate disruptive signals or events, or to change the physical environment in which a device functions, in order to trigger incorrect behavior in electronic equipment. This can be accomplished through direct contact with the target chip's external pins or using contactless methods such as heavy ion radiation or electromagnetic interference.

To target software applications and operating systems, a **Software-based fault injection** method is employed. This approach is appealing since it is inexpensive and does not require any additional hardware. The method has two kinds, depending on the time when a fault is injected. The Compile-time fault injection method alters the source code or assembly code of the system before the start of execution of the program. Whereas the Run-time faults are injected using special trigger events like interrupts to activate faults.

The **Simulation-based fault injection** is carried out in three different ways,

1. Digital Fault Injection Simulation
2. Digital Mixed Signal (DMS) Simulation
3. Analog Mixed Signal (AMS) Simulation

To begin, the Digital Fault Injection approach involves building a simulation model of the system under examination and faults using HDLs such as Verilog and VHDL.[52]. Second, for mixed-signal architectures that include both digital and analog components, novel techniques for fault injection at the system level are required. To perform the Digital Mixed Signal (DMS) simulations, the analog circuits are accurately modeled using various methods like Verilog-A, Verilog-AMS, Real Number Modeling (RNM), and SystemVerilog-based datatype EEnet. Finally, Analog Mixed Signal (AMS) simulates the precise schematics of analog circuitry with digital interfaces and tests the system for different analog faults like open, short, changing clock frequency and duty cycle and voltage and current spikes.

The final category is **Emulation-based fault injection**. This can be done in two ways, using hardware emulators or employing FPGAs [22]. Various EDA vendors provide emulators like the Palladium emulation platform offered by Cadence, the Veloce by Mentor Graphics, Zebu Server by Synopsys. These platforms support the verification of designs up to several billion gates. Despite the high execution speed and scalability of these emulators, the high price is a significant drawback.

The second option is the emulation of faults using FPGAs. There are two categories to perform emulation of faults on FPGA based on the type of implementation [21].

1. **Reconfiguration-based** technique where one fault can be configured at a time using partial or full FPGA reconfiguration [3],[56].
2. **Instrumentation-based** technique where additional hardware is integrated with the design to enable fault activation from an external host [41],[40].

In this study, the instrumentation-based emulation approach is used, where a custom fault saboteur is created to inject the fault into the design's gate-level netlist. Previously, this approach was used to examine the system's functioning. This thesis applies the principles of fault injection on the FPGA and safety-related techniques used in simulation-based fault injection to develop a system that emulates random faults in order to evaluate the system's safety on the FPGA and establish ASIL levels, as mentioned in section 2.1.1. Chapter 3 delves further into the two forms of fault injection, simulation-based and emulation-based, as well as the numerous strategies employed.

## 1.4. Thesis Contribution

The main goal of this thesis is to develop a fast and efficient platform to perform fault emulation on the FPGA for evaluating the designs for safety-related automotive applications to determine the ASIL level. There are several procedures involved throughout the development process of these ICs and even beyond once the chip is in the field. With the growing demand for faster chip time to market, there is a need to accelerate the development process. Furthermore, there are other factors such as labor cost, and resources available which impact the overall process. This endeavor aims to speed up the fault injection process in order to validate the functioning of Safety Mechanisms and provide diagnostics of fault classification to certify the product for ASIL.

### 1.4.1. Research goal

The primary goal of this research is to investigate the various techniques used in performing fault injection. Moreover, to evaluate these methods and formulate an architecture that helps to expedite this process. Currently, the fault injection activity is performed using extensive simulations which take days or even months. The main limitation of the simulation-based FI is the significant software overhead and availability of the resources. The simulation data provided by Analog Devices for one of their Battery Management Systems (IC) developed for ASIL D level showed the execution time for the fault injection activity to be 22 days. This is detailed in section 3.3. This work aims to bring this activity from simulation to the emulation world, building an architecture with minimal software overhead, and executing the fault campaign in less than 3 hours for a fault space of 1M.

### 1.4.2. Scope of the work

The state of the art revealed that FPGA-based fault emulation is one of the most feasible and cost-effective ways of performing fault injection. The fault injection activity mainly serves two purposes, firstly the diagnostic metrics of the fault campaign activity are required to provide evidence in order to certify the product before releasing it in the market. Secondly, evaluating the system's behavior in the field well during the development stage helps in designing the systems with the highest level of safety.

To perform the fault emulation on the FPGA requires three main components, firstly an automated flow that can map the design from ASIC to FPGA flow. Second, a hardware design that sits parallel to the Design Under Test (DUT) on the FPGA that can perform the fault injection and evaluation of the system on the FPGA, and lastly a software platform that can co-ordinate the fault campaign between the host PC and the FPGA. In this work, all these components are developed and tested. The motivation of the experiment is to evaluate the execution time of the fault campaign activity in comparison to the simulation-based method and achieve a minimum of 250x speed up.

### 1.4.3. Thesis Contribution

The outcome of this work is to develop a fault emulation platform that is fast and universal to perform the fault injection for a mixed signal system. There are three main components of this project,

1. First, a compiler was built to transform the gate-level netlist with a modular design for fault injection. This was accomplished using a Python-based platform and the ANTLR tool, which automatically deconstructs the netlist and implements the necessary changes without any manual intervention.
2. Second, the architecture of the emulation platform to perform the testing on the targeted design on the FPGA was developed. This design required to be fast enough to execute the functional and safety sequences with minimum overhead. This activity was accomplished in two stages: first, a simple architecture was developed, but had significant software overhead and was able to achieve a speedup of 43.2x; second, the limitations of this architecture were realized and improved in a final architecture with minimal overhead, allowing the time taken to execute the functionality of the DUT to be the largest time factor in the calculation of the fault campaign's execution speed and a speedup of 296x was achieved.
3. Finally, a software platform to efficiently communicate the campaign configuration from the host PC to the campaign manager on the FPGA was implemented, as well as record the outputs and execute the final fault categorization, which is the final outcome of this effort.

## 1.5. Report Organization

The rest of the report is divided into 9 chapters.
Chapter 2 describes the Functional Safety procedures used in the IC development process. It presents the ISO26262 standard for automotive applications and discusses the various ASIL requirements.

Chapter 3 discusses the various fault injection techniques used for evaluating the safety-critical integrated circuits. It discusses the various simulation-based methodologies and their limitations, as well as lays the groundwork for the emulation-based approach.

Chapter 4 introduces the FPGA-based Fault Emulation platform developed in this work. This chapter provides an overview of the system requirements and system specifications considered for this experiment as well as touches upon the flow of this project.

Chapter 5 establishes the preparation of the design for mapping onto the FPGA. It gives details about the gate-level synthesis and the netlist modifications with fault saboteurs.

Chapter 6 dives into the details of the two architectures experimented in this research. The various blocks of the system and their functions are explained. The main constraints of the first architecture's efficiency are underlined, and it is described how this is overcome in the second.

Chapter 7 focuses on the performance of the two architectures and a comparison of the two. It also provides the fault classification results for the second architecture which is the expected output of this project.

Chapter 8 summarizes the results of this work, discusses the simplifications made in order to expedite the activity without affecting the target goal of reducing the execution time of the fault injection activity, as well as mentions some of the limitations of the system.

Chapter 9 talks about the scope of future work. It illustrates how this project can be expanded in various areas.

# 2 Background on Functional Safety (FuSa) in IC design

The primary goal of Functional Safety (FuSa) in automobiles is to ensure the safety of human lives. The idea is to build systems that are reliable even in the face of unforeseen events. Initially, the concept of FuSa was applied to the development of electronics at a system level. But with increasing complexity and growing heterogeneous systems, its relevance has been observed at the chip development level. Today, automotive Original Equipment Manufacturers (OEM)s require safety-certified semiconductors to manufacture cars with high safety standards that gain the trust of their customers. This prompted EDA firms to see the tool development process as embracing the safety aspect. EDA companies like Cadence provide state-of-the-art solutions that cover parts of the safety spectrum, such as FMEDA (Failure Modes Effects and Diagnostic Analysis).

Moreover, the applicability of FuSa extends even beyond the IC development process. Electric vehicles (EVs), for example, employ a large number of Li-ion battery cells to achieve battery pack voltages in the hundreds of volts and capacities in the tens of kWh. These ICs necessitate safety measures not only throughout the design phase but also during the lifespan of the device in which they are placed. This draws focus on FuSa in the system on the fly.

In the following parts, the various safety aspects of the system and the standard followed during the IC design process is illustrated.

## 2.1. Introduction to ISO26262

Most of the cars today consist of complex technology, advanced software control, and mechatronics implementation. The risks posed by systematic and random hardware failures are also very high. Thus, to mitigate these risks, ISO26262 was introduced to provide universal guidelines for semiconductor development. The following are a series of standards provided by ISO26262 to achieve functional safety:

1. Offers a guide for the vehicle safety lifecycle and aids in designing tasks to be carried out during the lifecycle stages of development, manufacture, operation, service, and decommissioning.
2. Presents a risk-based strategy for determining integrity levels in the automobile industry [Automotive Safety Integrity Levels (ASILs)].
3. To eliminate unacceptable residual risk, ASILs are applied to identify which of the ISO26262 family of standards are relevant.
4. Specifies criteria for functional safety management, design, implementation, verification, validation, and confirmation measures
5. Describes the standards for customer-supplier relationships.

The process of developing a product is broken down into several stages, including the concept stage, product development at the system level, manufacture, operation, service, and decommissioning. Figure

2.1 shows a framework illustrating different activities during safety-related product development.

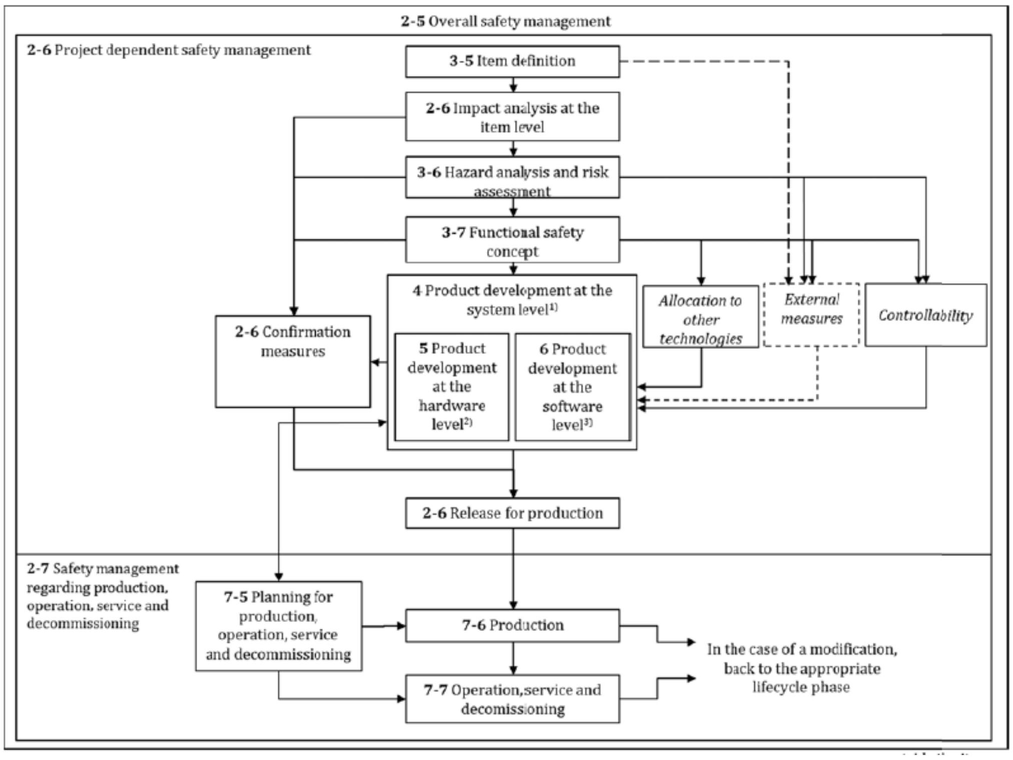

2-5 Overall safety management
2-6 Project dependent safety management
3-5 Item definition
2-6 Impact analysis at the item level
3-6 Hazard analysis and risk assessment
3-7 Functional safety concept
4 Product development at the system level1)
5 Product development at the hardware level2)
6 Product development at the software level3)
2-6 Confirmation measures
Allocation to other technologies
External measures
Controllability
2-6 Release for production
2-7 Safety management regarding production, operation, service and decommissioning
7-5 Planning for production, operation, service and decommissioning
7-6 Production
7-7 Operation,service and decomissioning
In the case of a modification, back to the appropriate lifecycle phase

**Figure 2.1:** ISO26262 Framework for safety-related product development [66]

The first stage of the safety lifecycle is the item definition, during which a comprehensive description of an item's functioning and risks is established. This is followed by hazard analysis and risk assessment, which evaluates the possibility of controllability, exposure, and severity of hazardous occurrences in relation to the item. Then, based on the safety goals specified by the preliminary architectural assumptions, a functional safety concept is constructed. Then comes the product development process at the system level, this is explained using a V-model shown in Figure 2.2. Finally, throughout the operation, service, and decommissioning phases, all requirements' specifications are confirmed, and instructions are given to assure functional safety during production. This phase runs in parallel with the others.

### 2.1.1. Automotive Safety Integrity Level

Automotive Safety Integrity Level (ASIL) is a risk categorization methodology based on the ISO 26262 standard for Functional Safety for Road Vehicles. The ASIL is determined in 4 steps as shown in Figure 2.3 The item definition of a function of component/block of a system is analysed using Hazard Analysis and Risk Assessment (HARA). This process requires the knowledge of ISO26262 framework and the teams's understanding of functional safety.

There are various activities performed in HARA such as,

1. Identify and categorize possible hazardous events, modes of operation, and environmental circumstances
2. Develop safety objectives in order to define a safe state
3. Indicate assumed safety goals and their associated ASIL
4. Summarize the customer's information on hazard analysis and risk assessment
5. To transform customer information in various formats into a standard format so that corresponding safety activities can be carried out
6. To provide a solid foundation for further safety operations for the product development

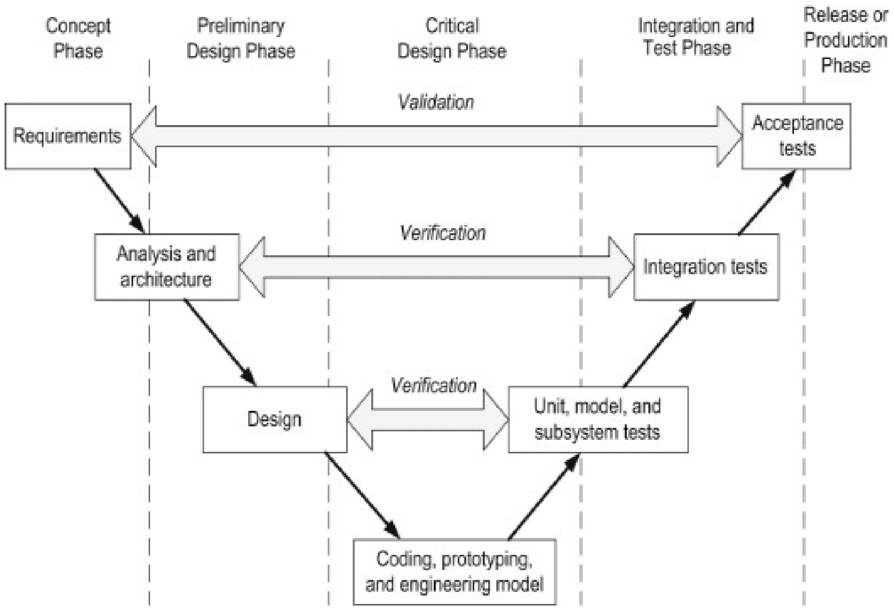

Concept Phase
Preliminary Design Phase
Critical Design Phase
Integration and Test Phase
Release or Production Phase
Requirements
Validation
Acceptance tests
Analysis and architecture
Verification
Integration tests
Design
Verification
Unit, model, and subsystem tests
Coding, prototyping, and engineering model

**Figure 2.2:** V-model shows the process of product development at system level [23]

HARA analyzes the three main criteria of ASIL, Severity, Controllability, and Exposure using the equation 2.1.

$$ASIL = Severity\ x\ (Exposure\ x\ Controllability) \tag{2.1}$$

Here, Severity is the measure of harm caused by the malfunction of the system to the people involved. Controllability determines the extent to which the vehicle is under control of the driver if certain operation of the system fails. The chance of a system failing or causing a dangerous condition is referred to as exposure. Figure 2.4[1] depicts the metrics used for determination. As shown in the figure, ASIL is classified into four classes: A, B, C, and D, with D being the highest safety level that demands more rigorous verification. Because there are no specific safety objectives, it is anticipated that hazards will be handled within the routine quality evaluation for the feature's functioning when a feature is classified as QM. The table 2.1 displays the minimum operating time (T) of a device "without faults" in relation to its ASIL level and table 2.2 defines the random hardware failure rates.

| **ASIL** | **Minimum operation time (T) without errors (hour)** |
|---|---|
| A | $1.2 \times 10^7 h$ |
| B | $1.2 \times 10^8 h$ |
| C | $1.2 \times 10^8 h$ |
| D | $1.2 \times 10^9 h$ |

**Table 2.1:** Minimum operation time without errors for ASIL classes[44]

| **ASIL** | **Random Hardware Failure Rate (FR)** | **FR in Failure in Time (FIT)*** |
|---|---|---|
| D | $10^{-8}/hour$ | 10 FIT |
| C | $10^{-7}/hour$ | 100 FIT |
| B | $10^{-7}/hour$ | 1000 FIT |
| A | Irrelevant | Irrelevant |

$* \ 1FIT = 10^{-9} failuresperhour$

**Table 2.2:** Random hardware Failure Rates (FR) in Failure in time (FIT) [44]

[1]https://www.linkedin.com/pulse/asil-automotive-safety-integrity-levels-ratings-animesh-sarkar

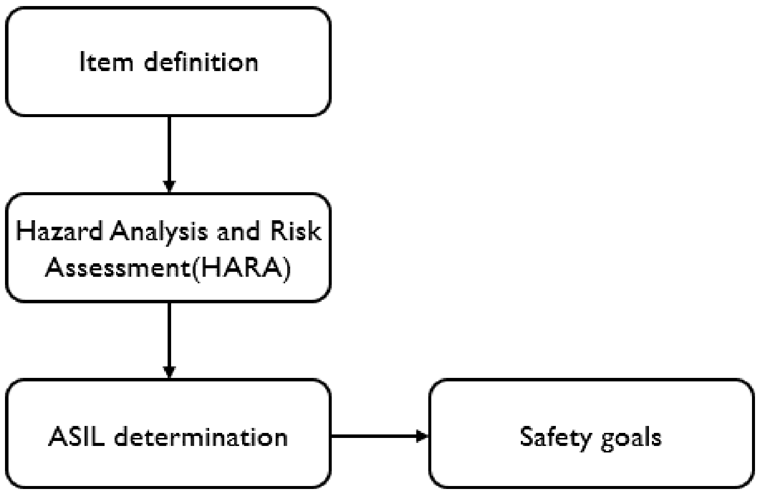


**Figure 2.3:** Steps in ASIL determination[44]

### 2.1.2. FuSa Verification process

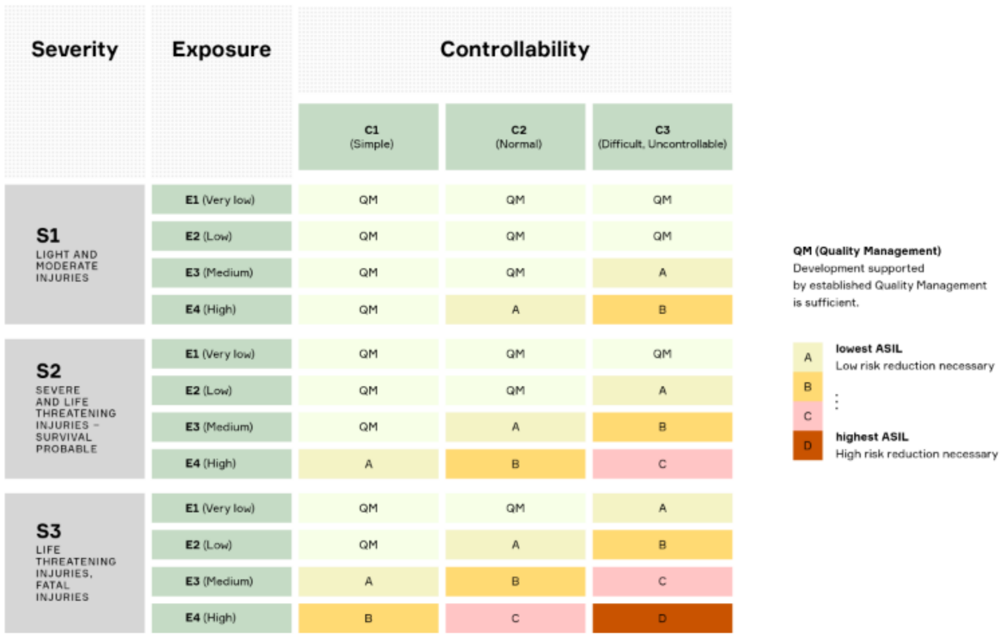
The verification process followed under ISO26262 mainly defines the "WHAT", "HOW" and "RESULTS". The verification plan helps to define "WHAT" is the strategy of the process. Here the objective, method of verification, pass/fail criteria, tools to be used, resources available and the regression strategy is laid out. Secondly, the verification specification defines "HOW" the process is carried out. There are checklists documented, simulation scenarios listed, test cases and objects are detailed, how different versions of work-product will be maintained is discussed, and environment and configurations are defined. Finally, all the results of the verification activities are documented. There are 5 types of verification being performed throughout the cycle.

1. Design reviews through inspection/walk-through which is a formal review process
2. Verification through engineering judgement
3. Simulation of various test scenarios
4. Bench Testing
5. Fault Injection

This project focuses on fault injection type of verification. The major distinction between fault injection activity for functional safety devices and other products is the verification of the system's accurate reaction. In general, fault injection is used to validate functional correctness. For example, if a defect exists, the fault injection activity should report incorrect output of the system and identify the location of the error; this aids in the design's improvement through techniques such as fault tolerance. These flaws might be random or systematic. However, they are assessed at the moment $t_0$ of the chip placement in the automobile.

The purpose of fault injection for functional safety goods, on the other hand, is to analyze the behavior of the chip placed in the automobile when it is in operation. For instance, consider a component shown in Figure 2.5 is deployed in a hybrid car, while being driven, a stuck-at fault(random) occurs in the fan-in of a flipflop X of the ALU logic at the Elementary sub-part level. This fault travels to the sub-part level as wrong data is being prepared by the ALU. The fault further disrupts the data generated by the CPU which results in wrong output of the Integrated Circuit at component level. This causes the actuator output to deviate and cause harm at the system level. Thus, a safety mechanism for the component is designed, and a safe state for the system is specified, to avoid and reduce the dangers produced by random errors occurred in the system on the fly. This implies that if something goes wrong in the IC, the safety mechanism is activated, and the system returns to a safe condition. Thus, the fault injection activity for functional safety product development is used for following purposes:

1. To evaluate the diagnostic coverage of safety mechanism



| Severity | Exposure | Controllability | | |
|---|---|---|---|---|
| | | C1 (Simple) | C2 (Normal) | C3 (Difficult, Uncontrollable) |
| S1 LIGHT AND MODERATE INJURIES | E1 (Very low) | QM | QM | QM |
| | E2 (Low) | QM | QM | QM |
| | E3 (Medium) | QM | QM | A |
| | E4 (High) | QM | A | B |
| S2 SEVERE AND LIFE THREATENING INJURIES – SURVIVAL PROBABLE | E1 (Very low) | QM | QM | QM |
| | E2 (Low) | QM | QM | A |
| | E3 (Medium) | QM | A | B |
| | E4 (High) | A | B | C |
| S3 LIFE THREATENING INJURIES, FATAL INJURIES | E1 (Very low) | QM | QM | A |
| | E2 (Low) | QM | A | B |
| | E3 (Medium) | A | B | C |
| | E4 (High) | B | C | D |

**QM (Quality Management)**
Development supported by established Quality Management is sufficient.

| | |
|---|---|
| A | **lowest ASIL** Low risk reduction necessary |
| B | ⋮ |
| C | |
| D | **highest ASIL** High risk reduction necessary |

**Figure 2.4:** ASIL determination metrics[31]

2. To evaluate the diagnostic time interval and the fault reaction time interval
3. Pre-silicon verification of safety mechanism with respect to the requirements, including its capacity to detect faults and control their effect (fault reaction)

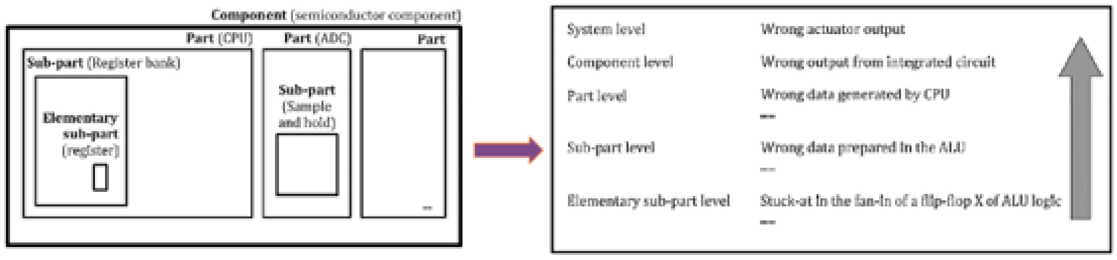

Component (semiconductor component)
Part (CPU)
Part (ADC)
Part
Sub-part (Register bank)
Elementary sub-part (register)
Sub-part (Sample and hold)
...
System level
Wrong actuator output
Component level
Wrong output from integrated circuit
Part level
Wrong data generated by CPU
Sub-part level
Wrong data prepared in the ALU
Elementary sub-part level
Stuck-at in the fan-in of a flip-flop X of ALU logic

**Figure 2.5:** Example of how fault travels from sub-part level to system level[32]

In the following parts of this section, certain terms defined in ISO26262 are explained.

**Type of failures**

A **Cascading failure** happens when a failure in one element creates problems in another. The element that is producing the issue is referred to as the root element. Figure 2.6 shows an instance of the occurrence of cascading failure. As seen in the figure, if some random fault occurs in Element A, and this element serves some functions of Element B, then the fault in Element A will trigger a failure in Element B.

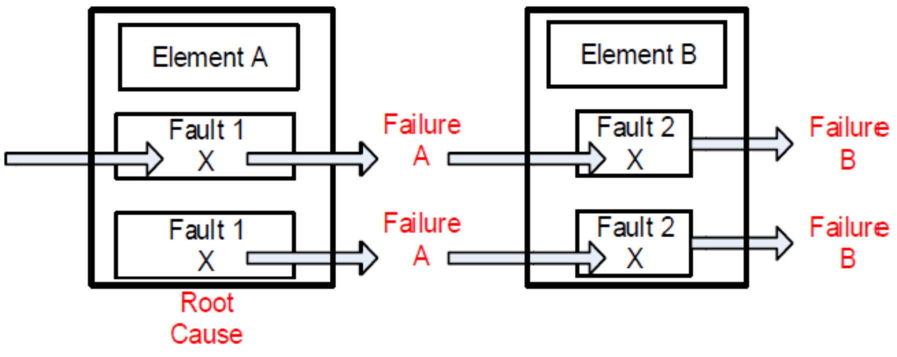


**Figure 2.6:** Cascading failure[1]

A **Common cause failure** occurs when two or more components of an item fail as a direct result of a single event that is either internal or external to all elements. Figure 2.7 shows an instance of common cause failure. Consider the two redundant blocks X depicted in the picture, and another Element B delivers some information to both of them. Consider that the output of both of these duplicate blocks has to be compared. Then, a failure in Element B will cause the identical fault in both the redundant blocks, and the outputs of both will be impacted equally.

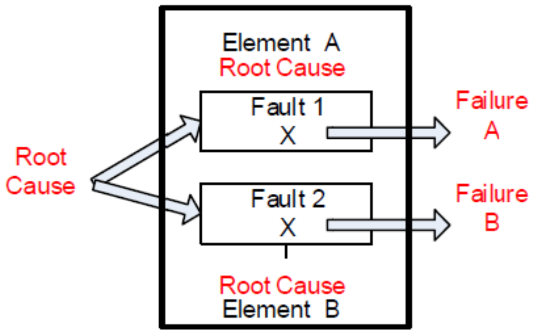


**Figure 2.7:** Common cause failure[1]

**Dependent failures**, as the name implies, are two or more failures that are dependent on one other. This does not imply that the product of all independent failures translates to the combined occurrences of dependent failures. The dependent failures have the capacity to exhibit themselves simultaneously or within a short time span to have a simultaneous impact. The common cause failures and cascading failures are also included in the dependent failures. Consider, Figure 2.8, blocks X and Y of Element A are dependent on each other, and in turn, facilitate the blocks X and Y of Element B. So any fault in blocks X and Y in Element A has the possibility to affect the blocks X and Y of Element B similarly even though these blocks are not connected to each other.

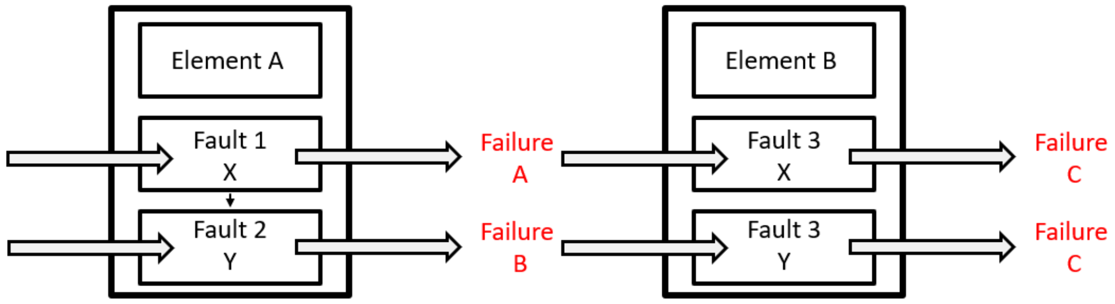


**Figure 2.8:** Dependent failure

**Independent failure** also known as Single-Point fault occurs when each defect in the system is not reliant on other faults. This is displayed in Figure 2.9.

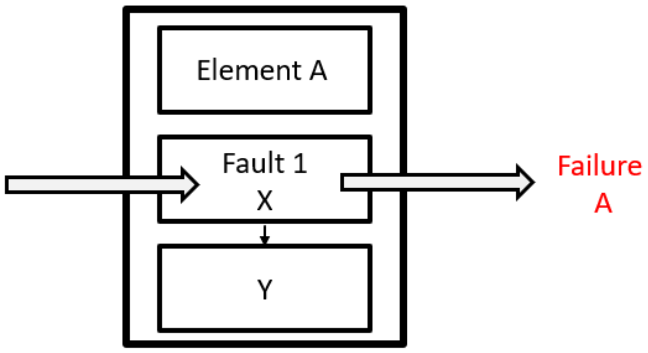


**Figure 2.9:** Independent failure

**Dual point failure** arises when two independent hardware faults exist in the system, resulting in a violation of the safety goal. Figure 2.10 illustrates this fault. Here the two blocks X and Y are independent of each other and cause two distinct faults in Element A.

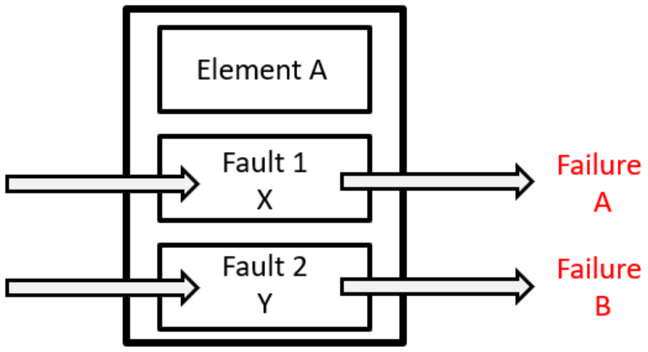


**Figure 2.10:** Dual point failure

**Random failure** can individually occur unpredictably during the lifetime of a hardware element. However, these failures can be modeled probabilistically. Random defects can arise for a variety of reasons, including heat stress, ageing, corrosion, and interactions with subatomic particles. Figure 2.11 depicts the connection of random failure with time.

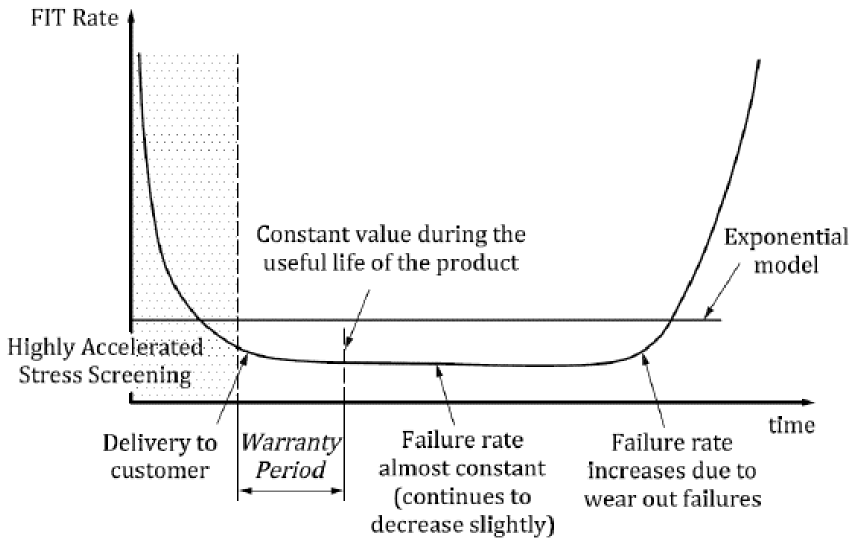


**Figure 2.11:** Semiconductor reliability curve[32]

**Systematic failures** occur during the product development process as a result of human error or error in the tool. These are deterministic faults that can be eliminated by enhancing the design and verification processes.

## Type of faults

The diagram below depicts all of the different types of hardware failures that can occur in a system. There are two types of failure modes: Safety-related failure mode and Non-safety-related failure mode. Non-safety related flaws do not breach any safety goals. There are five types of failures within the safety related failure mode: safe fault, multi-point fault, latent fault, single-point fault, and residual fault.

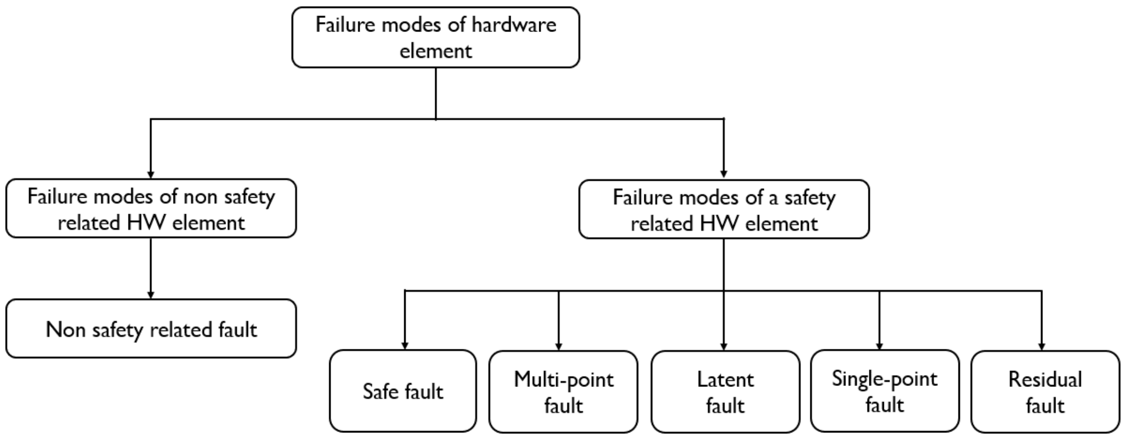

Failure modes of hardware element
Failure modes of non safety related HW element
Failure modes of a safety related HW element
Non safety related fault
Safe fault
Multi-point fault
Latent fault
Single-point fault
Residual fault

**Figure 2.12:** Type of hardware faults

A **Safe fault** is one that has little to no impact on the system's operation. These failures create a variation in functioning that is within the system's tolerances. A **Single-point fault** is a hardware fault present in the element that leads to safety violation and there is no safety mechanism is place to detect it. **Latent fault** is one that occurs in the safety mechanism (SM). However, a safety mechanism by itself does not violate any safety goals; often, a defect in SM happens in conjunction with a single-point malfunction. The next type of fault, as the name indicates, a **Residual fault** is created by a portion of the system yet directly violates the safety goal. There is no safety mechanism in place to regulate this unpredictable malfunction. Lastly, the **Multi-point fault** is a combination of two or more single-point faults or a single-point fault and a fault in the safety mechanism. The latter type of multi-point fault detection requires duplication of the safety mechanism. Figure 2.13 illustrates the occurrences of different faults and how it is detected by the safety mechanism (SM). The last type of fault which is not mentioned in the above figure is **Transient fault**. This issue arises for a brief period of time. If the duration of this defect exceeds the fault tolerance, it causes damage.

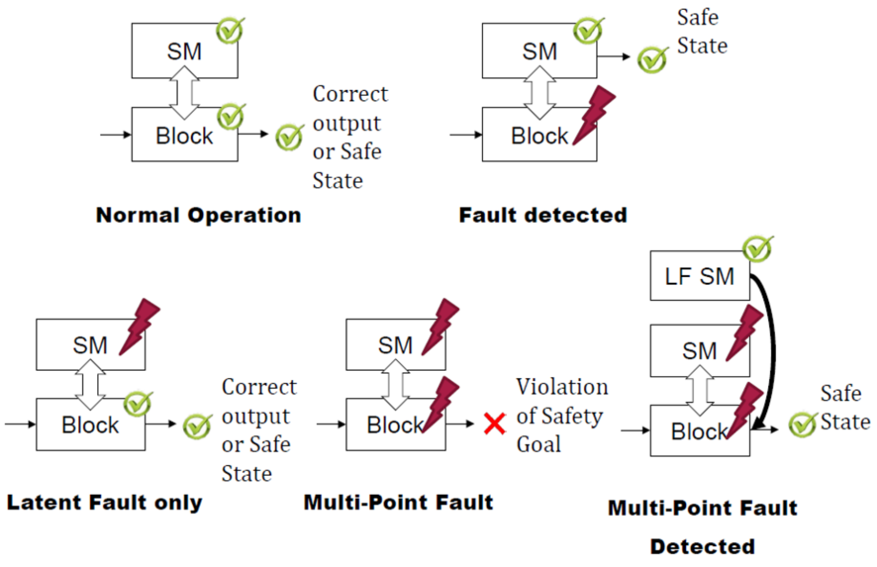


**Figure 2.13:** Fault occurrences and detection[32]

The two important faults are single-point faults and latent faults that cause the major issues. The performance metrics required for these faults to qualify different ASIL classes is shown in table 2.3.

| | ASIL A | ASIL B | ASIL C | ASIL D |
|---|---|---|---|---|
| Single-point fault metric | Irrelevant | $\geq$ 90% | $\geq$ 97% | $\geq$ 99% |
| Latent fault metric | Irrelevant | $\geq$ 60% | $\geq$ 80% | $\geq$ 90% |

**Table 2.3:** ISO26262 fault performance metrics [44]

### Failure mode and Safe mode

The **Failure mode** is simply the state where something goes wrong in the system. When a failure occurs, the system enters **Safe mode**. This is a state where unreasonable risks are mitigated.

### Safety Mechanism

A random failure can occur at any moment in any aspect of the system owing to a multitude of factors such as environmental conditions such as light radiation flipping a memory fit or thermal stress changing the state in a key register. As a result, several safety mechanisms are developed to limit the impact of these unpredictable hardware errors. Some of these approaches are addressed in [44], including Error Correcting Codes (ECC), Dual and Triple Modular Redundancy (DMR/TMR), Lock step, DMR Hardened Flip-Flops, Built-in Self Test (BIST), and Cyclic Redundancy Checking.

In Figure 2.14, the effect of faults with and without safety mechanisms is demonstrated. When an element/item fails, the system continues to function normally until the fault tolerance is reached. The fault is not detected for a specific time span known as the fault detection time interval. During this time, various safety sequences are executed regularly to capture the presence of fault. Once the fault is identified, the system returns to a safe state; the time necessary for this is referred to as the fault reaction time interval. The fault detection and reaction time are combined to form the fault handling time. In the absence of safety mechanisms, the minimal time span between the occurrence of a problem in an item to a potentially dangerous period is known as the fault tolerant time interval.

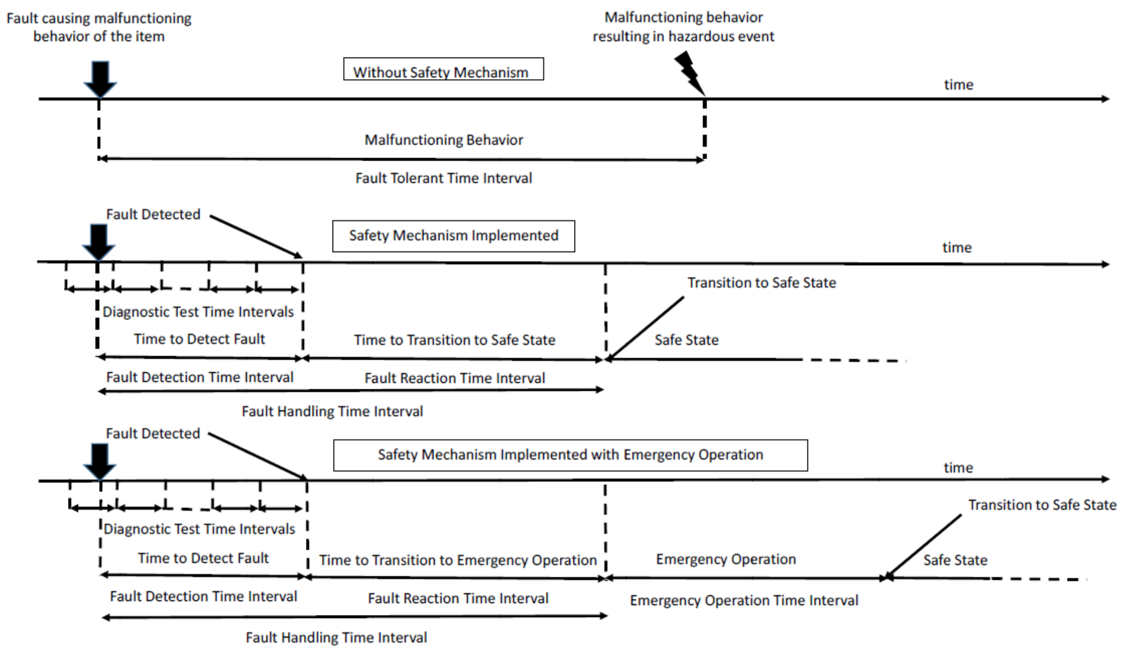

Fault causing malfunctioning behavior of the item
Malfunctioning behavior resulting in hazardous event
Without Safety Mechanism
time
Malfunctioning Behavior
Fault Tolerant Time Interval
Fault Detected
Safety Mechanism Implemented
time
Transition to Safe State
Diagnostic Test Time Intervals
Time to Detect Fault
Time to Transition to Safe State
Safe State
Fault Detection Time Interval
Fault Reaction Time Interval
Fault Handling Time Interval
Fault Detected
Safety Mechanism Implemented with Emergency Operation
time
Transition to Safe State
Diagnostic Test Time Intervals
Time to Detect Fault
Time to Transition to Emergency Operation
Emergency Operation
Safe State
Fault Detection Time Interval
Fault Reaction Time Interval
Emergency Operation Time Interval
Fault Handling Time Interval

**Figure 2.14:** Safety related time intervals[1]

**Fault classification**

To evaluate the fault campaign, there are mainly two outputs observed, namely Functional Output (FO) and the Checker Output (CO). The functionality of the system can be determined from the functional output, and the checker output gives information about the faulty behaviour of the system. For instance, if the output of a temperature sensor is to be measured and the threshold of the system under test is set to a particular range. Now, if the sensor output deviates from the set range, the CO sets high and FO shows the amount of deviation. Using the two outputs FO and CO, any fault can be classified into four types:

- Unobserved Undetected (UU)
- Unobserved Detected (UD)
- Dangerous Undetected (DU)
- Dangerous Detected (DD)

When neither the functional nor the checker outputs are affected by the fault, the UU case occurs. UD indicates that the functional output is undetected but the tool output is set. This situation is not harmful since the user is aware that some functionality is not behaving properly. Thus, UU and UD can be categorised as safe faults. The following one, DU, is the most perilous since the functional output is detected whereas the checker output is not. In this test case, the user will be unaware that the system is failing and posing a risk. The verification engineer's purpose is to disclose the majority of DU situations and indeed provide proof that the design meets all the safety requirements. The last case DD is equally dangerous but it has been recognized, so any mishaps caused by it can be avoided.

The fault coverage can be calculated as:

$$Fault\ coverage = \frac{DU}{DD + DU + UD + UU} \tag{2.2}$$

## 2.2. Example: Battery Monitoring System in Electric Vehicle

The transition to Electric Vehicles (EVs) in the automotive sector is accelerating. Lithium-Ion batteries are a main component of electric vehicles. The presence of a considerable number of these batteries is responsible for a range of risks such as electric shock, thermal events, and poisonous gas release, necessitating the implementation of a Battery Management System (BMS) system with high functional safety standards. Furthermore, because lithium-ion batteries have changing performance depending on environmental conditions, active monitoring of voltage, current, and temperature is essential to operate the system efficiently. Due to the criticality of the BMS system, it requires a higher ASIL validation.

The BMS continuously monitors varying parameters in different operating modes and takes the necessary actions to prevent cell degradation. In [54], some of the BMS functions are mentioned below,

1. Cell voltage and temperature monitoring
2. Wake-up and sleep management
3. Cell balancing
4. Network management
5. Fault management and isolation detection
6. Battery charge monitoring

Once the functionality of the BMS system is known, the system is evaluated using HARA to identify all the critical points causing hazards. These hazards are categorized in terms of severity, exposure, and controllability. Consider the table below illustrating the HARA activity for battery charge monitoring.

| **BMS Function** | Battery charge monitoring |
|---|---|
| **Possible failure modes** | Failure in the charge monitoring functionality |
| **Operational Scenarios** | Vehicle plugged in for charging while occupied |
| **Hazard Impact (Vehicle Level)** | Toxic gas leak from parked car |
| **Severity class (S)** | S3 |
| **Possible consequences** | Vehicle occupants lose consciousness and are at risk of getting fatally intoxicated as a result of the poisonous gas buildup within the passenger cabin. |
| **Probability class (E)** | E4 |
| **Use cases** | Travel on the highway and make a charging stop. |
| **Controllability class (C)** | C3 |
| **Controllability Assumptions** | Occupants are oblivious of the danger. They may abandon the car owing to tiredness, but they are unlikely to be aware of the origin of the problem and therefore the essential solution. |
| **ASIL** | C |

**Table 2.4:** HARA example for BMS system

After determining the ASIL grade, safety goals for single or multiple hazardous incidents are developed. These safety goals signify the top-level safety standards. They result in functional safety standards for each hazardous event in order to avoid an erroneous risk. Table 2.5 presents the functional safety requirements,

| **SG Description** | **ASIL** |
|---|---|
| Avoid exposing humans to high voltage electrical energy (HV E/E) | B |
| Avoid Vehicle Thermal event that cannot propagate through the vehicle or damage to safety-critical components | C |
| Prevent the hazardous emission of poisonous fumes from the High Voltage Battery (when the car is plugged in and charging). | C |

**Table 2.5:** BMS Safety goals

Based on these requirements, the functional safety concept is drawn and the system architecture is developed considering the ASIL grading. Table 2.6 displays a few functional safety requirements for the BMS system.

| **Functional Safety Requirement** | **ASIL** |
|---|---|
| Any cell voltage that falls below a certain threshold is considered an undervoltage fault | C |
| Any cell voltage beyond a certain threshold must be identified as an overvoltage malfunction | C |
| Any cell temperature that falls below a certain threshold is considered a temperature defect | C |
| Any cell temperature that exceeds a certain threshold is considered an excessive temperature malfunction | C |

**Table 2.6:** FSRs for BMS system

Consider the above-mentioned first function of the Battery Management System (BMS) IC, which continually checks cell voltage and current. Consider the following system design: three ADC converters and an SPI serial protocol to convey digital voltage and current to the host. These ADCs are constructed utilizing majority voters logic, which requires that the majority of redundant blocks provide the same outcome. Random faults may be inserted into the ADCs and analyzed by performing multiple safety sequences to test this system. This safety check must occur concurrently with the routine monitoring of the battery cells' voltages and currents.

# 3 State of the art in Fault Injection

Any electrical system may be examined for two types of behavior: positive and negative. Non-critical systems are often evaluated for positive behavior, which implies that the functionality is checked for correctness within the expected range of behavior. However, the risks associated with safety-critical ICs, such as the battery management system IC in electric vehicles, are so significant that these ICs also require negative testing. Furthermore, evaluating these chips solely before they are distributed to consumers is insufficient. They must be constantly monitored for their behavior. For example, if a random failure occurs in a system when the vehicle is in motion, it is important to determine how fast the system can detect the fault and react to a failure. In such a case, the system should identify the error before it causes excessive harm and return the system to safe mode, without affecting the functionality, in a controlled manner. As can be observed, the criticality of the system dictates the path of system verification throughout IC development. This leads us to the 'WHY' fault injection is required. Fault injection is an excellent strategy that allows the engineer to test the design at a system level in every possible scenario. This helps in evaluating system behavior when the chip is in operation. Performing fault injection activity at the development stage assists in gauging chip functioning in the field, estimating the severity of the hazard, and reflecting on architectural improvements if required that may be vital when designing ICs with high safety requirements.

The fault injection processes are performed in a variety of methods to target specific parts of the system and evaluate its capacity to detect, respond to, and recover from defects in a safe and controlled manner. The four types described in [57] explains 'WHAT' needs to be done during the fault injection activity,

1. **Stimulus-based fault injection**: This includes injecting faults into the system on purpose by modifying input data, changing internal state, or triggering error conditions.
2. **Structural fault injection**: This process involves introducing faults by restructuring the source code, changing the execution routes, or interfering with memory access.
3. **Environmental fault injection**: This type of FI entails exposing the system to situations that may create defects such as temperature extremes, voltage fluctuations, or electromagnetic interference.
4. **Timing fault injection**: This includes changing the timing of system events, such as introducing delays or race situations, in order to create failures.

During the functional verification phase, the design is examined for all structural flaws. This procedure is primarily concerned with the design's functional robustness. The test cases are developed to examine the SOC for all of the intended scenarios and completeness, as well as to determine the system's code and functional coverage. Secondly, the Environmental faults in electronic circuits are more akin to voltage spikes, representing external disturbances that can disrupt the normal operation of a circuit. However, circuit designs are meticulously crafted to proactively mitigate these environmental faults rather than introducing them intentionally. The clock monitors, supply monitors to monitor voltage and current variations, and temperature sensors to detect temperature extremes are used for monitoring environmental faults. Thirdly, in the context of Timing Analysis, engineers scrutinize extreme timing conditions, particularly at the boundary between analog and digital components in mixed-signal systems. Parasitic extraction further enhances the validation process by assessing factors such as skews and device variability. These meticulously designed circuits aim to minimize timing variations and complexities, ensuring the absence of inherent timing violations. Instead, the primary concern revolves around random faults that could potentially lead to logic failure. To counter these challenges, Automatic Test Pattern Generators (ATPG) are deployed, operating at much higher frequencies than the circuit's typical operational frequency. This proactive approach serves as a safeguard against unforeseen disruptions in logic integrity. Lastly, the Stimulus-based fault injection approach is investigated in this project. This form of fault injection activity can be carried out in a variety of ways. The parts that follow will provide an in-depth discussion of 'HOW' fault injection can be implemented utilizing different strategies.

## 3.1. Methods of performing Fault Injection

In this section, the two common ways of performing the fault injection to evaluate the system for safety are discussed. The Simulation-based and Emulation-based methods displayed in Figure 3.1 are detailed in the following parts.

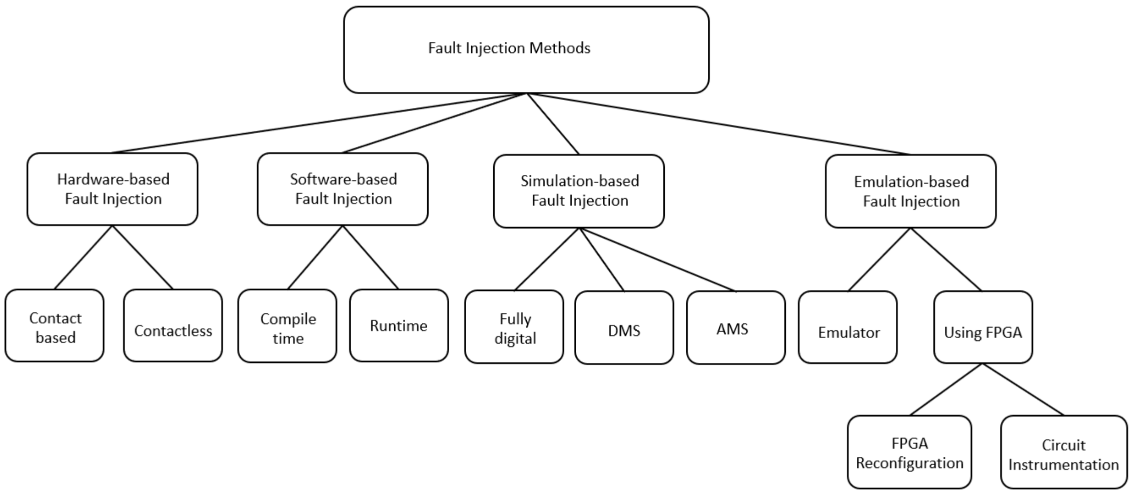


**Figure 3.1:** Various methods used for performing Fault Injection

To manufacture ASIL-certified goods, the ISO26262 standard requires a rigorous verification procedure to be completed. The main goal is to demonstrate that the design is immune to random failures in the system while the car is in motion and that it meets all ISO26262 standards. Because functional safety is not confined to product inspection during development but extends even while the system is in use, traditional fault modeling approaches such as ATPG and BIST are no longer applicable. Because these methodologies can only be used to evaluate the system before the IC is released to the market. Thus, to complete the safety verification, fault campaigns are run where defects are introduced, propagated, and categorized. The fault classification is then utilized to provide the necessary safety metrics. The overview of a typical fault campaign is displayed in Figure 3.2.

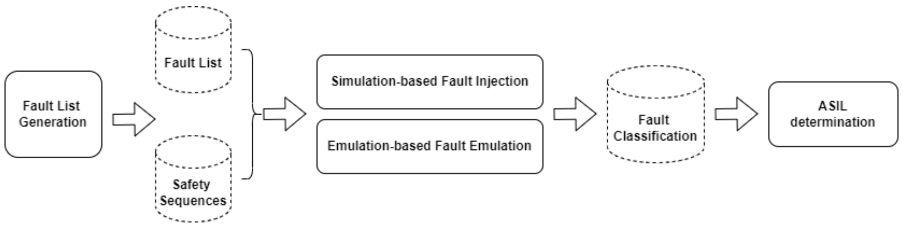


**Figure 3.2:** An overview of a Fault Campaign

The fault campaign is run in three stages,

1. **Fault injection phase** where a single or dual fault is injected at a particular site in the design.
2. **Execution phase** where different functional stimuli and safety mechanisms targeting different parts of the design are executed.
3. **Fault classification phase** uses two sorts of outputs, namely the functional output and the checker output, to categorize the defect as harmful, unharmful, found, or undetected.

Currently, mainly simulation-based fault injection technologies are used in the industry. However, when the complexity of the SOCs increases and the fault space grows into millions, this technique becomes impractical for evaluating the whole fault space. As a result, several alternative approaches such as formal verification and tactics such as statistical sampling are used to acquire the diagnostic coverage proof necessary for product certification with the intended ASIL rating. Regardless of these solutions, the time and money required are significant. The many state-of-the-art fault injection strategies are explored in the following section.

### 3.1.1. Simulation-based Fault Injection

The fault injection simulation is carried out in three different ways,

1. Digital Fault Injection Simulation
2. Digital Mixed Signal (DMS) Simulation
3. Analog Mixed Signal (AMS) Simulation

The Digital Fault Injection method entails creating a simulation model of the system under consideration and faults using HDLs like Verilog, VHDL [52]. The simulation of the fault campaign is performed using various tools like Xcelium Fault Simulator by Cadence or the Unified functional safety verification platform by Synopsys. Obtaining DFI findings involves several processes, including rating the test cases, fault instrumentation, good simulation, fault simulation, and report generation, as demonstrated in Figure 3.3 with an example of vManager Safety Client used to execute DFI.

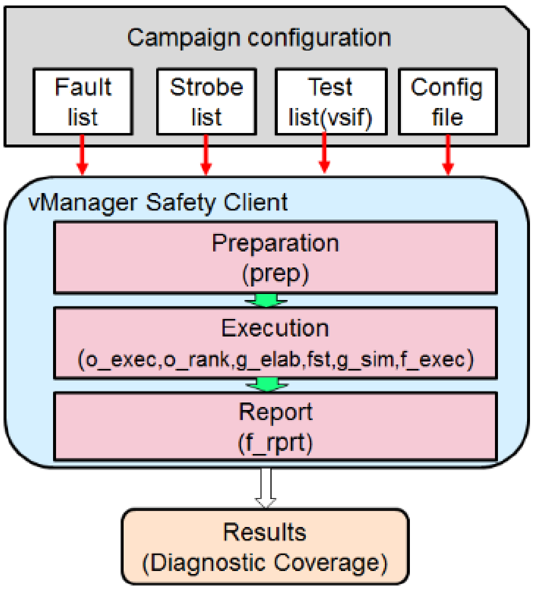


**Figure 3.3:** Digital Fault Injection flow using vManager Safety Client

Before running a fault campaign, various parameters need to be configured shown as Campaign configuration in the diagram. The Fault list contains the definitions of all the faults to be injected into the design. The Strobe list provides information about the observation points. The Test list demonstrates the testcases that need to be executed during the campaign along with their priority order. Lastly, the Config file helps in defining the environment of the fault campaign. Once all these files are ready, the campaign is performed in three steps, preparation, execution, and report. During the preparation stage, a node is selected one at a time and the fault is injected. Then various functional and safety sequences are executed and the behavior of the system is recorded. First, these sequences are run when no fault is present in the system and this is regarded as the golden run. Now, faults are injected one by one and the outputs are recorded again. Based on these results, the faults are classified and a diagnostic coverage is determined as discussed in section 2.1.2.

Furthermore, most systems nowadays are mixed signal in nature, comprising both digital and analog components, necessitating the use of innovative approaches to execute fault injection at the system level. To perform the Digital Mixed Signal (DMS) simulations, the analog circuits are accurately modeled using various methods like Verilog-A, Verilog-AMS, Real Number Modeling (RNM), and SystemVerilog-based datatype EEnet. In [2], the author presents modeling the analog circuits using Verilog-A and the netlist generation flow for fault injection. More complex analog circuits can be designed using RNM modeling. In [26], a SystemVerilog-based real number model for a Voltage-Controlled Oscillator (VCO) is presented to improve simulation performance. The authors also compare VCO models utilizing RNM and Verilg-AMS modeling. The SystemVerilog model achieves significant improvements in simulation run time while maintaining acceptable accuracy.

While the DFI concentrates on the digital portion of the SOC, the analog portion consists of models designed using EEnet, Wreal, and RNM modeling with a certain degree of accuracy. Thus, to verify the actual analog schematics, Analog Mixed Signal (AMS) simulations are performed. AMS, like Digital Mixed Signal Simulations, is concerned with the analog domain. To verify the system, detailed schematics of analog circuitry with digital interfaces are employed. Because of the nature of the defects, AMS fault injection is more difficult than fully automated DMS fault campaigns. In the analog realm, there are many more faults than in the digital domain. The most common ones are open, short, changing clock frequency and duty cycle and voltage and current spikes.

As the complexity and size of the design grow, considering the chip time to market, it is not feasible to perform fault injection for 100% of the fault population. Thus, various strategies are applied to target different aspects of verification and are discussed in the following parts.

To limit the number of fault nodes evaluated for performing fault simulations while maintaining the quality of outcomes, a technique known as Statistical Sampling is employed. In a Statistical Sampling Fault (SFI) campaign only a random portion of the total fault population is simulated, and the results are extrapolated from a subset to a larger set. In [37], an approach is proposed to quantify these results by rigorously evaluating the margin of error and confidence interval for SFI campaigns. This paper presents a mathematical formula to derive the subset of faults that are representative of the total fault population and error margin using the following equation 3.1 and 3.2 respectively,

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{3.1}$$

$$e = t \times \sqrt{\frac{p \times (1-p)}{n} \times \frac{N-n}{N-1}} \tag{3.2}$$

where n is the sample size, N is the total fault population, p is the estimated proportion of a fault resulting in a failure, e is the margin of error, and t is the critical value corresponding to the confidence level.

The confidence level is the likelihood that the precise number is inside the error interval. A confidence level of 90%, 95%, or 99% is selected based on the ASIL level required as explained in table 2.3.

Using the equations above, the authors demonstrated that for a given fault population of size 150 trillion, the sample space can be reduced to 30,000 injections ( reduction in experimental time by 7 orders of magnitude), leading to a 0.57% margin of error with a 95% confidence level, or a 0.1% margin of error with t=0.3464, accounting to a 27% confidence level, or a 1% margin of error with t=3.4641, which is almost a 100% confidence level.

### 3.1.2. Digital Fault Injection Emulation

It is clear from the preceding discussion that Simulation-based fault injection has several constraints, most notably the execution speed of the fault campaign. To accelerate this effort, emulation-based fault injection was investigated. The design is modeled on hardware, defects are physically inserted into the DUT, and the system is subsequently assessed. This results in significant hardware acceleration over software simulations. The next parts go through the various approaches for emulating the fault injection activity.

The Emulation-based fault injection can be done in two ways, using hardware emulators or employing FPGAs. In article [22], the emulators provided by various vendors are discussed. For example, the Palladium emulation platform is offered by Cadence. This emulator can be used to build, allocate, run, and debug during the design cycle with a maximum speed of 4MHz. The Veloce emulation platform is another one that is offered by Siemens's Mentor Graphics. This platform supports the verification of designs up to several billion gates. Another EDA player, Synopses also offers an emulation system called Zebu Server. This platform claims to be the fastest with a single rack system capacity of 1.2 billion gates and a multi-rack system supports up to 10 billion gates. Despite the high execution speed and scalability of these emulators, the high price is a significant drawback.

The second option is the emulation of faults using FPGAs. There are two categories to perform emulation of faults on FPGA based on the type of implementation [21].

1. **Reconfiguration-based** technique where one fault can be configured at a time using partial or full FPGA reconfiguration.
2. **Instrumentation-based** technique where additional hardware is integrated with the design to enable fault activation from an external host.

The Serial Fault Emulation (SFE) approach in which each faulty circuit is emulated separately, implemented using the reconfiguration technique to evaluate single stuck-at faults in digital systems was attempted as early as 1996 [9]. In [11], the Serial Fault Emulation method was improved by two speed-up techniques, first, all independent faults are identified and emulated parallelly, second, to inject multiple faults, extra circuitry is added during the design phase to inject faults. Furthermore, the multiple bitstream generation problem is addressed by simply shifting the contents fault injection chain to activate one fault at a time[28]. Another approach to dealing with multiple FPGA reconfiguration, Dynamic partial reconfiguration (DPR) is presented in [24]. Dynamic partial reconfiguration (DPR) is a feature that allows only a piece of the FPGA to be reconfigured without shutting down the FPGA or deleting the whole of the configuration. This article discusses two approaches of DPR implementation. A partial bitstream is created based on the difference between a reference bitstream and the target design in a difference-based DPR flow. This method is helpful for making minor modifications to the FPGA's setup, such as modifying memory content and lookup-table entries. The module-based DPR flow divides a design into distinct modules, and faults are injected into a smaller sub-circuits HDL file, thus reducing the compilation time. Many approaches with the Reconfiguration method are discussed in [3],[56],[4],[60],[16], however, the Instrumentation-based approach outweigh the former in terms of speed of execution.

In [41],[40], an Instrumentation-based approach that challenges the previous method was presented. Here, an autonomous emulation strategy is discussed and three techniques of circuit instrumentation are compared in terms of speed of emulation. First, a mask-scan approach where an additional flipflop is integrated with all the sequential elements of the design as shown in Figure. 3.4. These additional flipflops indicate where fault is inserted.

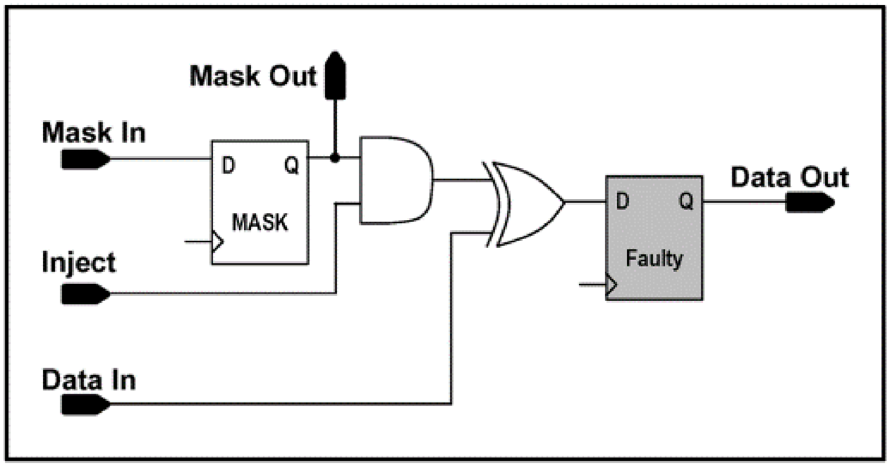


**Figure 3.4:** Flip-flop replacement for the Mask-Scan emulation technique [40]

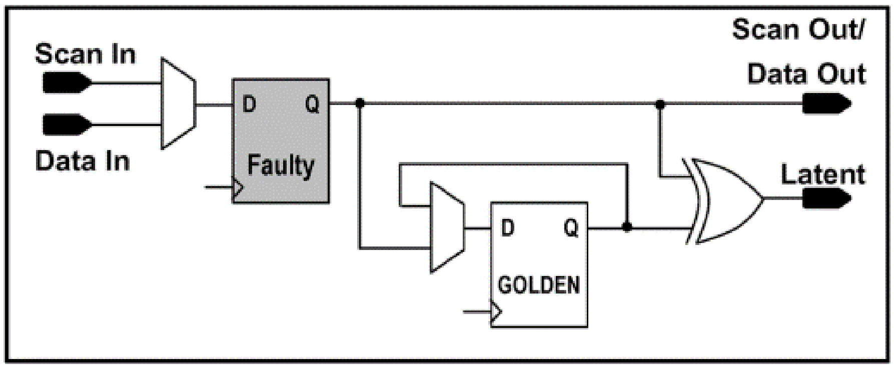


**Figure 3.5:** Flip-flop replacement for the State-Scan emulation technique[40]

In the second method, a state-scan technique is applied where all the sequential elements of the design are connected together to form a scan chain, and the state of each element is recorded in RAM. The circuit replacing the original flipflop in this technique is shown in Figure. 3.5 During fault emulation, these states are controlled and evaluated.

The last practice was to replace every flop in the design with a time-multiplexed technique as shown in Figure. 3.6.

In this diagram, two flip-flops are utilized to alternately run the flawed circuit and the golden run, resulting in time multiplexing. A third flip-flop serves as a failure mask, and a state flip-flop is utilized to prevent restarting the emulation from scratch every time. This approach is much quicker than the previous ones since it detects fault effects without rerunning the entire testbench. This improves the performance of transient fault emulation by some order of magnitude when compared to the simulation(1300us/fault)[41]. Furthermore, [40] provides improved performance and comparison of all three methods, mask scan, scan state and time multiplexed. The performance reported mask scan technique is $4.1\mu s/fault$, for state scan roughly 11 $\mu s/fault$, where as for time-multiplexed technique is 0.57 $\mu s/fault$.

In [39], yet another unified environment is introduced and compared against FPGA partial reconfiguration method. The proposed architecture can inject exhaustive fault sets (129.75 million faults) in an extremely short time (fault rate in the order of $\mu s$/fault) at a 400% flip-flop and 320% LUT additional area cost. Partial Reconfiguration, on the other hand, can inject subsets of faults (10000 faults) at medium rates (fault rate in the order of 0.1s/fault) at an extra area cost of 163% in flip-flops and 217% in LUTs.

In an Instrumentation-based solution, two code modification techniques, Saboteur and Mutant are

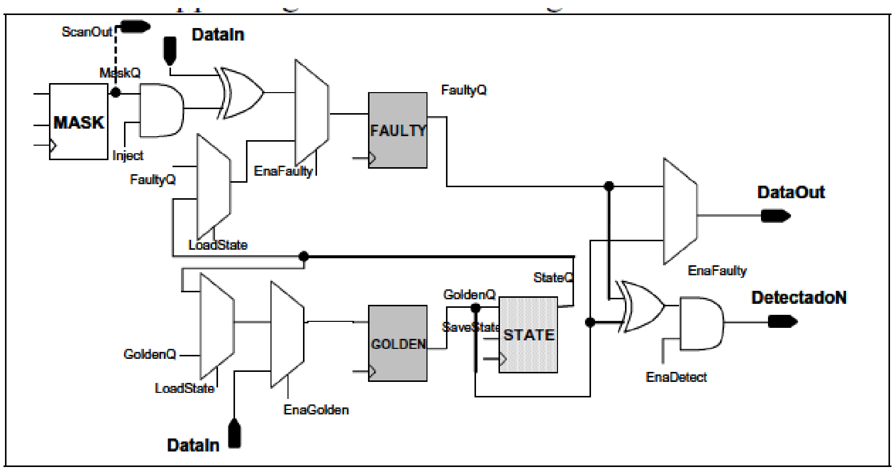
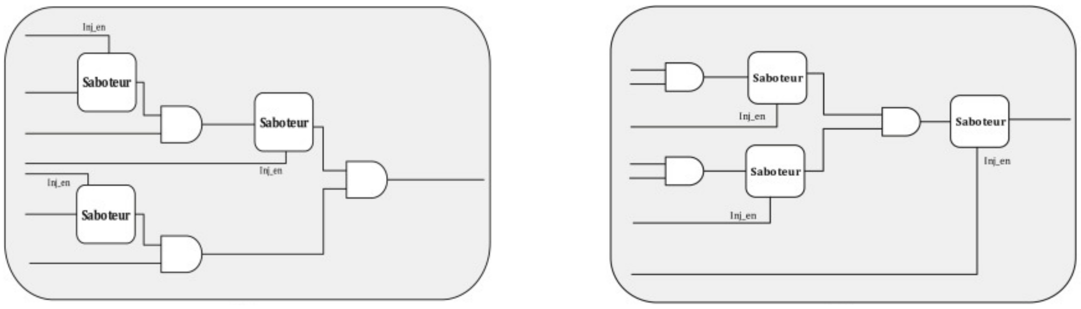


**Figure 3.6:** Instrument for time multiplexed [41]

employed [53]. For the first technique, an extra circuit for injecting faults known as saboteur is integrated with the design [21]. All saboteurs have an enabling signal, they remain inactive during the normal operation of the system and faults are activated when the relevant fault site enable signal is triggered.



**(a)** Adding saboteur to the input parts

**(b)** Adding saboteur to the output parts

**Figure 3.7:** Adding saboteurs at fault sites [21]

In the second technique, when the fault injection phase is active, a mutant is administered in place of the original component; otherwise, the mutant is inactive. The biggest disadvantage of this strategy is that the mutant must be recompiled for each new configuration. This is why, in comparison to Saboteurs, this practice is less commonly utilized.

With technology nodes shrinking and SOCs becoming more complex, the engineers are urged to conduct more rigorous pre-silicon verification of the design. This has further encouraged the fault injection activity to analyse the behaviour of the ICs in the presence of random and intentional faults. Apart from system complexity, the security aspect of the IC also plays an important role for certain applications. To evaluate the hazards caused by the system malfunction in the presence of random faults, real-time scenarios are created and emulated on FPGA platforms. Identifying the failure modes at an early stage of design helps in producing safer chips. To perform fault injection on FPGA, several environments setups are available. In [12], an environment architecture demonstrated to perform fault campaign on FPGA is shown in Figure. 3.8. In this figure, there 4 main components of the architecture are

1. Fault List Manager is in charge of producing the list of faults to be injected into the system, as well as the input data and any designer indications (e.g., the most crucial areas of the system). Furthermore, it is in charge of putting strategies in place to reduce the size of the fault list by consolidating comparable issues or deleting faults.
2. Fault Injection Manager is at the heart of the Fault Injection environment, organizing the selection of a new fault, its injection into the system, and the monitoring of the subsequent behavior.

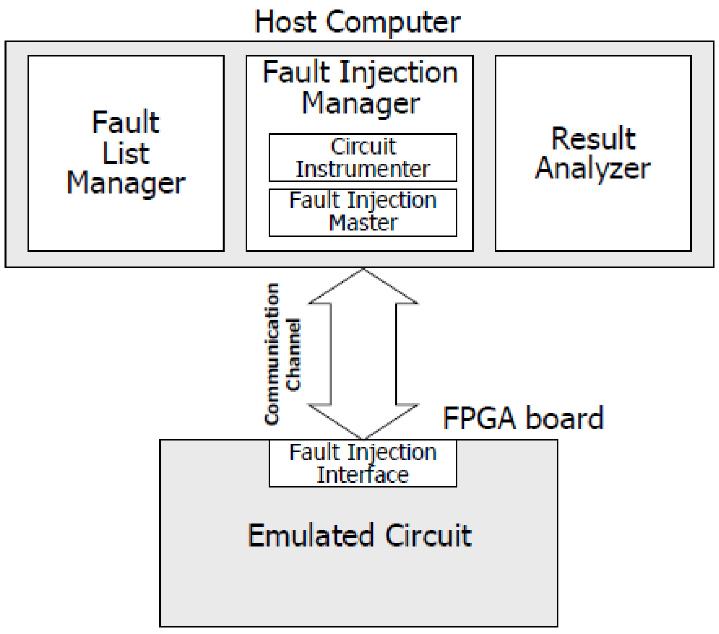


**Figure 3.8:** Overview of Fault Injection environment [12]

3. Result Analyzer analyzes the output data produced by the system during each fault injection experiment, categorizes problems based on their impact, and generates statistical statistics.
4. FPGA board consisting of Design Under Test (DUT) and fault injection interface.

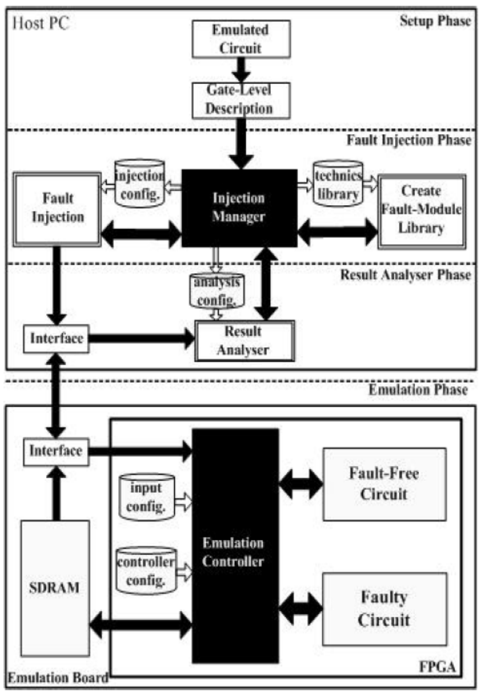
The Speed-up factor of upto 60x was reported against gate-level simulation-based Fault Injection approaches. Another similar architecture exhibited in [68] is shown in Figure. 3.9a and Figure. 3.9b explains the fault emulation process.

The components of the architecture like Injection Manager, Result Analyzer work similar to what was described in the previous example. Additionally in this example, Emulation controller is implemented in the FPGA sends the input vectors and controller vectors to the Fault-Free Circuit and Faulty Circuit. The performance of the FITVS approach is compared to fault simulation using Modelsim for same design. FITVS has a performance of around 8$\mu s$/fault, whereas Modelsim simulation takes 1.04s/fault. Various architecture are attempted to improve the execution speed. Related architectures to evaluate Single Event Upset (SEU) for digital system are demonstrated in [34], [30], [48], [67], [13]. [34].

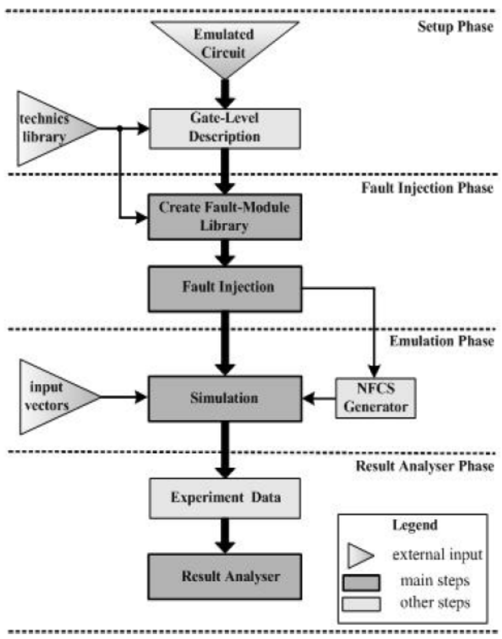
The state of the art methodology for FPGA-based Fault Emulation is presented in [33]. The Meta model architecture demonstrated in this article is shown in Figure. 3.11.

First, DUT is modified with saboteurs using a framework shown in Figure. 3.10. The saboteurs technique which enables fault injection is similar to what was discussed previously in this section. To improve the area overhead due to saboteur, the saboteurs are introduced only for the logic where fault injection is targeted in GL granularity, remaining logic is maintained at RTL level. Then, synthesis is performed maintaining the GL/RTL mixed design.

The fault injection flow is described in Figure. 3.12. This flow's three key components are the Fault List, Fault Controller, and Fault Analyzer. The fault injection occurs automatically in the majority of the architectures stated above, and the results are accurate, however, the fault campaign is done manually. The fault campaign procedure is likewise automated in this research. There are three types of fault campaigns that can be used: statistical fault injection, which tests a subset of total faults, direct fault injection, which allows the user to inject faults into a specified set of signals, and exhaustive fault injection, which tests all potential fault locations. The fault campaign output is transferred to the Post-Processor, where the faults are classified into three types: silent, latent, and failure. A silent fault is one that does not propagate to the primary output. The latent fault goes to the internal important registers rather than the principal output. When a fault is detected at the primary output, failure ensues.



**(a)** Architecture of FITVS



**(b)** Fault emulation process

**Figure 3.9:** Architecture of FITVS and Phases of Fault Emulation[68]

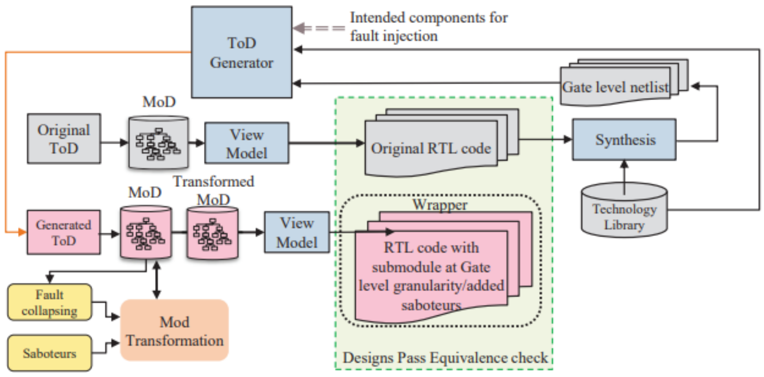


**Figure 3.10:** Mixed register transfer/gate-level design generation flow [33]

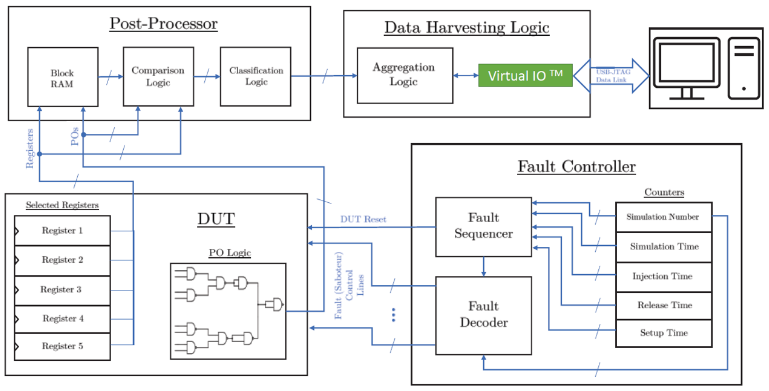


**Figure 3.11:** Architecture at block-level[33]

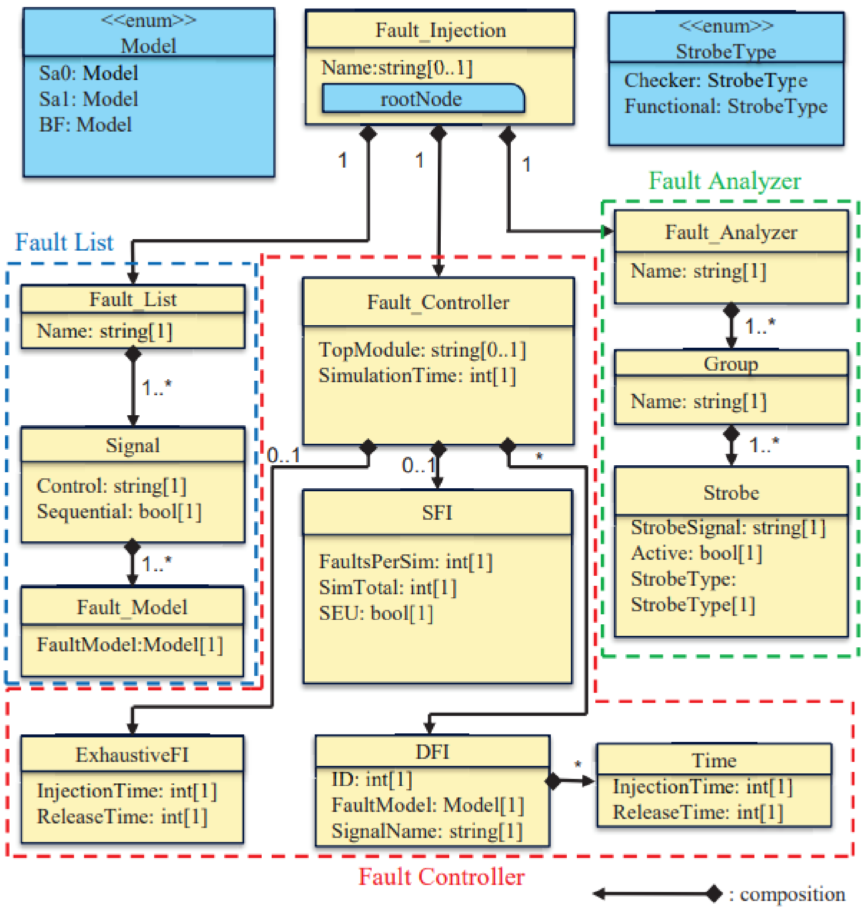


**Figure 3.12:** Fault injection metamodel [33]

All of these outputs are preserved in block RAM and can be accessed by the host outside of the FPGA.

Every fault has a state preserved in the block RAM; as the number of faults increases, this becomes a constraint in terms of scalability; so, another approach known as on-the-fly emulation was considered to enhance this. This approach replicates the DUT, thus there are now two DUTs, one for Fault Injection DUT(FIDUT) and second is Golden. The fault is injected into the FIDUT, and the outputs of both DUTs are compared on the fly during the emulation. This resolves the block RAM constraint. The experiments were performed on RISCV processor consisting of 5 CPU stages, Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB). The performance metrics of on-the fly emulation against simulation are displayed in table 3.1.

| **CPU stage** | **Faults Injected** | **Emulation campaign** hh:mm:ss | **Emulation runtime** ss:ms | **Simulation campaign** hh:mm:ss | **Overhead** |
|---|---|---|---|---|---|
| IF | 12799 | 02:27:50 | 64.054 | 42:20:56 | 17.28 |
| ID | 16803 | 03:13:50 | 84.094 | 67:17:54 | 20.91 |
| EX | 27616 | 03:33:57 | 138.213 | 168:52:18 | 47.57 |
| MEM | 1298 | 00:38:10 | 6.491 | 09:23:56 | 14.82 |
| WB | 493 | 00:27:14 | 2.462 | 03:51:16 | 8.56 |

**Table 3.1:** On-the-fly emulation technique runtime values [33]

This example gives an excellent model for conducting fault campaigns, however, it cannot be used to determine ASIL levels, as stated in section 2.1.1. This method also ignores the safety measures implemented in the design of safety-critical automotive ICs. In an ideal world, there is no golden DUT, and defects are only diagnosed with the assistance of safety mechanisms integrated into the design, which is also not addressed in the preceding example. A detailed explanation of this project's contribution is provided in section 1.4.3.

## 3.2. Selected method for Fault Emulation

In the preceding section, two emulation approaches were presented: one that used FPGAs and the other that used emulators offered by EDA firms. The latter is a costly alternative, and the resource constraint for this activity utilizing simulation also applies here, as the number of cores accessible is determined by the number of licenses acquired by the company. As a result, it is clear that FPGA-based fault emulation is one of the most practical methods for performing fault injection. Currently, Xilinx FPGAs on the market are significantly less expensive than the tool licenses given by EDA companies. Despite the fact that the FPGA-based approach requires one-time non-recursive efforts to translate the design implemented in the ASIC flow onto the FPGA, the speed-up achieved by performing the activity on FPGA outweighs the one-time effort.

## 3.3. Benchmark

To evaluate and compare the performance of the FPGA-based fault emulation platform to be developed in this project, the simulation performance of one of the Battery Management System (BMS) ICs was used. Table 3.2 presents the various parameters involving in performing the Digital Fault Injection (DFI) using the Xcelium Fault Simulator (XFS) by Cadence.

| **Description** | **Unit** | **Value** | **Value** | **Thesis Target Value** |
|---|---|---|---|---|
| Max cores available for Digital Fault Injection | | 160 | 50 | |
| Realtime duration of a functional pattern, including classification | s | $10^{-2}$ | $10^{-2}$ | $10^{-2}$ |
| Number of faults | | 640K | 640K | 640K |
| Single DMS simulation duration | s | $3 \times 10^2$ | $3 \times 10^2$ | |
| Campaign duration (w/o overhead) (overhead ∼ 50%) | s | $6 \times 10^5$ | $1.92 \times 10^6$ | |
| Campaign duration | days | 6.94 | 22.22 | < 2hours |

**Table 3.2:** DFI performance of Battery Management System IC using Cadence Xcelium Fault Simulator

The simulations described above were carried out using statistical sampling, with just 5% of the overall fault population simulated. This demonstrates that when the fault space expands, simulation of the complete population becomes nearly impossible due to the limited amount of resources available for a certain activity. This emphasizes the need to develop a platform that not only enables hardware acceleration of the fault campaign activity but also eliminates resource dependency for this activity.

# 4 Fault Emulation System Development

## 4.1. Project Overview

As seen in the preceding section, FPGA-based emulation is one of the most viable and cost-effective ways to do fault injection. The Xilinx FPGAs are significantly less expensive than the tool licenses provided by EDA vendors. A medium-sized FPGA with a million logic gates costs around 5,000 euros, but the licenses by EDA firm cost many tens of thousands of euros. Furthermore, the FPGA-based solution is not only less expensive, but it also eliminates the resource availability limitation. Thus, FPGA-based fault emulation provides the best hardware acceleration to speed up the fault injection activity.

Previously, fault emulation on FPGA has been attempted to evaluate the functionality of the system in the presence of faults. However, not much work has been done to evaluate the system for safety. This work focuses on the design strategies and the platform required for testing safety-related designs, especially for automotive applications. For example, consider Battery Management Systems (BMS) used in electric vehicles. To manufacture extremely safe chips for the automotive industry, it is necessary to examine the behavior of the ICs in the event of a random failure assuming the vehicle is in motion. The host system of the crucial ICs continually analyzes its behavior by running the safety sequences every few milliseconds and takes the appropriate action between the safe time intervals. Moreover, these safety checks need to be performed in parallel to the normal operation of the system. The fault injection activity enables engineers to examine system behavior in the face of a defect on the fly during the IC development phase, facilitating the design of ICs with high safety requirements and a low risk of failure. The safety metrics produced from the fault injection activity serve as proof of the FMEDA for the product and are critical in certifying the IC.

Before delving deep into the system's development process, it is critical to grasp the needs and specifications. Some of the system requirements and specifications for this task are provided below.

## 4.2. System requirements

Now that the various system tasks of the fault emulation platform are identified, there are several requirements that this system must adhere to. While designing the architecture for this project, the following key parameters were prioritized: speed of execution of the fault campaign, cost-effectiveness of the hardware, and system scalability. Keeping these parameters in mind, the system requirements are detailed below,

1. Should be faster compared to the traditional simulation-based FI. The DFI simulations roughly take 22 days for a fault population of size $\sim 600K$ faults with 50 tool licenses. The system should be able to complete the same task in roughly $\sim 3hrs$.
2. Results should be 100% accurate compared to the results obtained from the simulation-based FI.
3. Should be complete in terms of all the types of fault that will be tested (SPF, DPF and TF).

4. Should be scalable. (The system should be feasible for 100% of the fault population). For this experiment, the maximum fault population size of 1M faults is assumed.
5. Should be easy to integrate or fully replace with the existing digital mixed signal (DMS) verification flow. The main constraint in completely replacing the DMS simulations with the fault emulation system is the modeling of Analog circuits. However, most of the analog models can be accurately mapped on the FPGA using SystemVerilog datatype Wreal, which has a digital nature.

## 4.3. System specifications

FPGAs are the most feasible solution that provides hardware acceleration to perform digital fault injection. They are cheap compared to the EDA providers which makes them an economical solution. Moreover, FPGAs also solve the resource constraint and the activity can be performed independently. However, when the design grows considerably complex, there is the possibility to integrate multiple FPGAs and extend the fault injection activity but this comes at a cost of some initial setup like synchronization between the multiple FPGAs. Thus, designing a medium-sized fault campaign of size 1M faults on a single FPGA board that only costs a few hundred euros is the most feasible implementation of the fault injection activity. Mentioned below are the system specifications to fulfill the system requirements,

1. The FPGA should be large enough to accommodate the design of roughly 1M gates.
2. The design after modifications to map onto the FPGA must be equivalent to the original design to ensure that the functionality is intact.
3. The design of the fault saboteur should be robust to run all types of fault campaigns (SPF, DPF, TF).
4. The two main elements programmed on the FPGA are the Design Under Test (DUT) and the logic performing the fault injection. The architecture of the system managing the fault campaign should be designed with optimal area in order to accommodate large DUTs.
5. The fault injector logic should be designed to execute the injection process faster than the real-time execution of the functional and safety sequences in order to achieve the maximum total fault campaign execution speed.

Keeping the system requirements and specifications in mind, the overview of this work demonstrating the various steps involved is introduced in Figure 4.1.

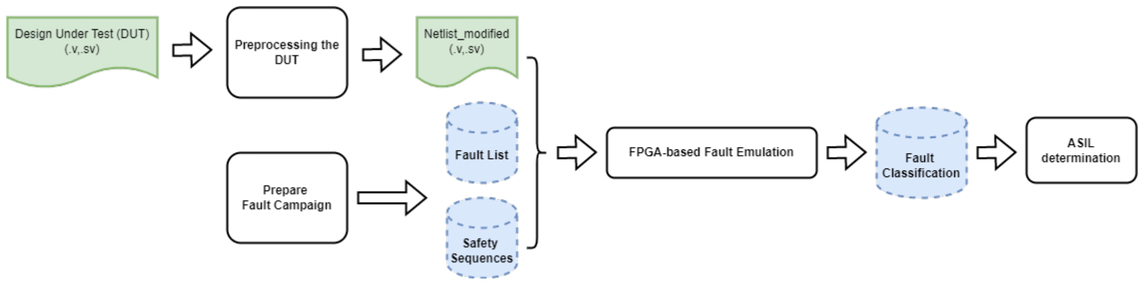


**Figure 4.1:** An overview of Fault Emulation using FPGA

There are four key tasks in this experiment. To begin, the Design Under Test (DUT) must be prepared for testing; second, the fault campaign must be configured; third, the fault campaign must be carried out; and lastly, the fault classification is done and the results are used to calculate the safety metrics. Based on the tasks, three action flows are devised, namely the pre-processing flow, the hardware design flow, and the software design flow and are outlined in the following sections.

## 4.4. Pre-processing Flow

The first step in the fault emulation activity is to get the design ready for testing on the FPGA. The primary objective should be to ensure that the design does not lose functionality throughout the revision. The steps involved in this flow are displayed in Figure 4.2. This activity consists of two parts, mainly the Gate Level Synthesis and secondly, modification of the generated netlist using Python framework.

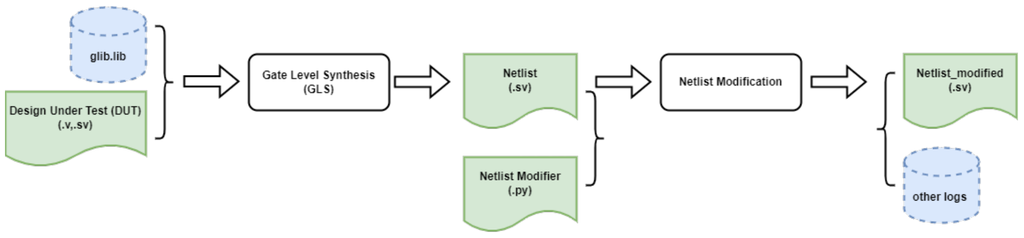


**Figure 4.2:** Process Flow

To begin with, synthesis is conducted to convert the Verilog RTL design into a technology-specific Gate Level netlist. The Cadence Genus synthesis tool was used for this. Secondly, the netlist cannot be directly programmed onto the FPGA. it requires provision to create faults in the nodes of the gates in the netlist. Thus, a Python-based compiler is developed using ANother Tool for Language Recognition (ANTLR) parser that recognizes the structured Verilog or SystemVerilog netlist, decomposes the various elements of the code, and then translates the desired parts.

After modifying the netlist, it is critical to review the functioning. A logic equivalence check (LEC) is done for DUT versus the revised netlist for this reason. This task is carried out using a Conformal tool from Cadence which uses adaptive proof technology for checking RTL-gate comparisons. This tool analytically assesses all of the situations in both the cases and outputs if the two are equal. The design is now FPGA-ready for testing.

### 4.4.1. Hardware Design Flow

Once all design modifications have been completed and the netlist is ready for testing, a platform is necessary to run the fault campaign. The Hardware Design Flow displayed in Figure 4.3 depicts the many components that are programmed on the FPGA. The primary goal of this flow is to develop an architecture of the Host Debug Interface (HDI) slave surrounding the DUT that controls the fault injection, runs the safety sequences, and records the output of the system behavior. While designing the HDI slave, three primary features were kept in mind: the speed of the fault injection in the DUT, minimal software and hardware overhead while executing the functional and safety sequences, and the minimum area consumption to allow maximum space for the DUT. For this experiment, a Xilinx Arty A7 100T FPGA was chosen. As a result, the supporting FPGA design tool Vivado was employed to synthesize the design and generate bitstream to be programmed on the FPGA. The design of the HDI slave and its interfaces with the DUT are explained in detail in Chapter 6.

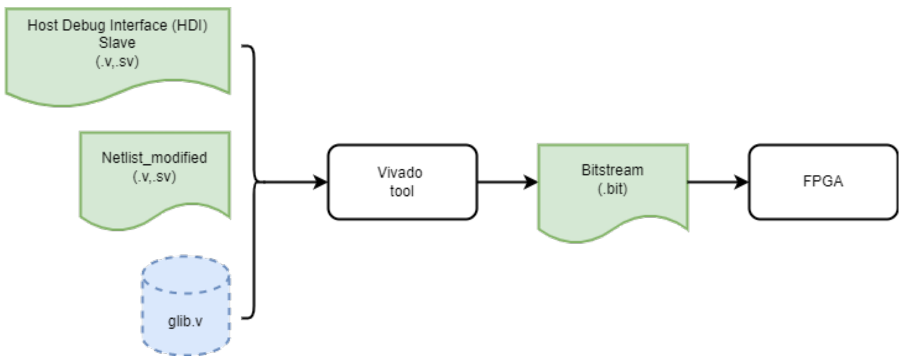

Host Debug Interface (HDI) Slave (.v,.sv)
Netlist_modified (.v,.sv)
glib.v
Vivado tool
Bitstream (.bit)
FPGA

**Figure 4.3:** The Hardware Design Flow

### 4.4.2. Software Design Flow

The last component of this project is the Software Design Flow. Once the hardware is programmed on the FPGA. In order for the HDI slave to run the fault campaign on the FPGA, it requires campaign data such as fault type, type of campaign, fault activation and deactivation time, and safety sequences. This project experiments with two sorts of faults: Stuck-at-1 (SA1), where a node is permanently trapped at logic high, and Stuck-at-0 (SA0), where the node is locked at logic low. And three types of campaigns are used: Single-Point Fault (SPF), where only one fault is active at a time in the system, Dual-Point Fault (DPF), when two faults are active at the same time, and Transient Fault (TF), where the fault is present for a short period of time and then vanishes. The Host Debug Interface (HDI) master as shown in Figure 4.4 is a Python program designed to transmit this information from the host PC to the FPGA using a serial communication protocol. In this case, UART serial protocol is exercised. Here, two channels of USB to UART are used, one for transmitting the fault information from the Host PC to the FPGA and the second one for receiving the functional outputs back from the FPGA to the Host PC. Finally, the results are utilized to carry out the fault categorization and establish the appropriate ASIL-level safety metrics.

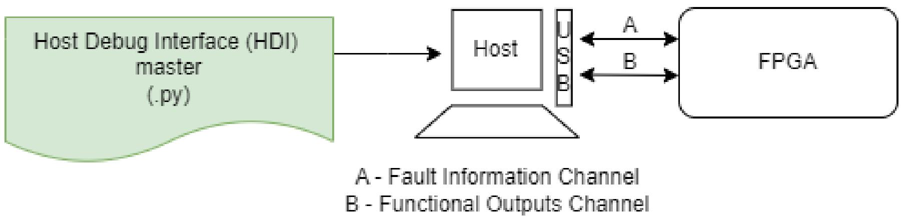

Host Debug Interface (HDI) master (.py)
Host
USB
A
B
FPGA
A - Fault Information Channel
B - Functional Outputs Channel

**Figure 4.4:** The Software Flow

In the next chapter, detailed information on the processing of the Design Under Test (DUT) is provided. Because the project required extensive preparation of DUT, hardware, and software design, some simplifications were permitted by the stakeholder during the preprocessing stage to speed up the development process which are discussed in the chapter 5. However, these do not affect the performance of the outcome.

# 5
# Pre-processing of DUT

To accomplish fault injection in the netlist on FPGA, a fault must be physically created in the design. Thus, several considerations must be made while mapping the design to be tested on the FPGA. As previously described in section 4.4, the design must first be synthesized and then revised to map onto the FPGA. The many processes done throughout the synthesis process and netlist translation are detailed in this section. To begin, some simplifications were made during synthesis to ease the mapping of gate-level netlist onto the FPGA. These consequences are discussed in further detail below.

## 5.1. Performing Gate-Level Synthesis

Gate-level synthesis is the process of converting a high-level hardware description into a netlist of logical gates and flip-flops. The Cadence Genus synthesis tool [25] was employed for this purpose. It optimizes the design for factors like area, power consumption, and timing constraints. The process involves mapping the design to a target technology library, which includes standard cells.

Given the project's complexity, some simplifications were made during the GLS with the future phases of this experiment in mind, particularly while translating the netlist onto the FPGA with custom-designed fault injectors. These simplifications were evaluated before hand and realized that they do not impact the project's performance or completion. Based on these criteria, the four simplifications are as follows.

### 5.1.1. Simplifications during the Gate-Level Netlist

1. **Netlist Flattening** - To translate the netlist with fault injector, the Python-based compiler is required to recognize the Verilog structure of the netlist and modify the required parts. Typically, the synthesized netlist is hierarchical in nature, which makes it slightly difficult to deconstruct the netlist at different abstraction levels and then modify the input nodes. Thus, to simplify the development of the Python script, the Netlist Flattening constraint was exercised during the design GLS. However, it must be noted that this does not affect the execution performance of the campaign.

2. **No DPT option** - Design For Testability (DFT) is primarily used to facilitate efficient and effective testing of integrated circuits and electronic systems. DFT techniques, such as scan chains and built-in self-test (BIST) structures, enable the generation of test patterns automatically. ATPG tools create a set of test vectors that can be applied to the circuit to detect and diagnose faults, ensuring thorough testing of the design. However, adding DFT during the GLS requires additional effort while mapping the netlist on the FPGA. The FPGA does not have an inbuilt scan flops like the other flops like D flipflop. These need to be custom-designed and included while mapping the netlist onto the FPGA. As this does not affect the motivation of this project to evaluate the execution speed of the fault campaign, the DFT option was not included during the design's GLS.

3. **No clock-gating** - Clock gating is a power-saving technique used in ASIC design to reduce dynamic power consumption by selectively disabling the clock signal to specific circuit elements when they are not in use. This technique helps conserve power in situations where parts of the circuit do not

need to operate continuously, such as in low-activity portions of the design or during idle states. However, FPGAs are often unsuitable for clock gating, unless particular design strategies are used, if the clock gating is done straight out of fabric on the FPGA, the FPGA synthesis tool will issue a warning about the skew injected. Furthermore, at very high frequencies, this skew becomes challenging to control. Thus, in this work, no clock gating technique was implemented in any part of the Design Under Test to ease the transformation of design from ASIC flow to FPGA.

4. As a consequence of the previous simplification, no faults were injected into the Clock tree.
5. Lastly, the Technology library during GLS was restricted to a limited number of cells mainly inverter, AND, OR, XOR, and DFF to ease the burden on the netlist modification script using Python.

When transitioning designs from the ASIC flow to the FPGA, a design attribute occurs during RTL synthesis owing to the implementation of the Safety feature. This is seen below, and it demonstrates how this attribute is emulated on the FPGA.

## 5.2. Skew between clock-trees

Designing safety-critical integrated circuits entails more than just ensuring functioning. The system must be designed considering the occurrence of random failures in the environment and the system's response in their presence in mind. The system must be capable of recognizing these random errors and reverting to a safe condition within the time frame specified before an unmanageable danger occurs. Because of this, certain design strategies are used to ensure that the system operates safely even in the face of random defects. One of the ways is described in the next part, and it is explained how this strategy followed in the ASIC flow is replicated on FPGA.

### 5.2.1. Artificial skew generation FPGA

The ISO26262 standard describes the many design strategies that are used in a SoC to make it Safety Compliant. Some of the approaches employed in safety-compliant devices include redundancy, self-testing, signal monitoring, voltage supply monitoring, and watchdog timers. The essential component of these devices is redundancy. Redundancy is employed in a variety of ways in car equipment. Many safety-compliant devices include redundancy techniques such as lockstep, ECC, CRC, and checksum [35][18][46].

Redundancy is the installation of extra components in addition to those required for vital functions to boost the system's dependability and availability. The M-out-of-N (MooN) scheme is one of the redundancy techniques explained in [19]. An M-out-of-N (MooN) system has N identical components and operates on the premise that if at least M out of N components perform properly, the system is error-free. Triple Modular Redundancy (TMR), which is actually a 2oo3 system, is an example of this displayed in Figure 5.1.

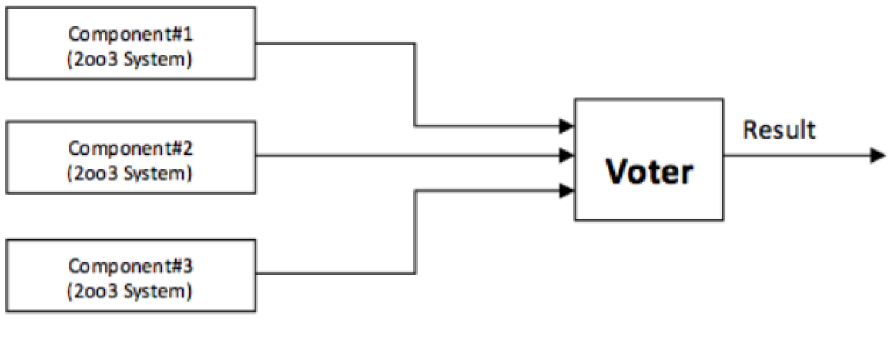


**Figure 5.1:** Example for Triple Modular Redundancy (TMR)[19]

In TMR, if at least two (majority) of the three components perform well, the system is deemed functioning. The design under test in this study consists of logic duplication, one known as the functional block and the second as the safety block, as illustrated in Figure 5.2.

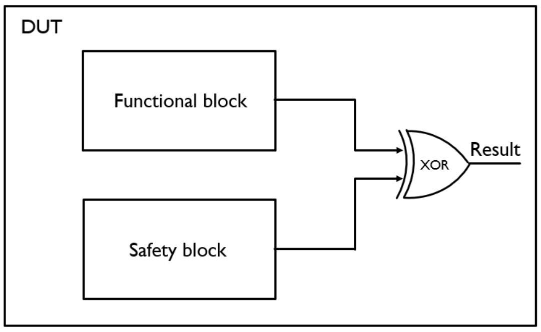


**Figure 5.2:** Example for duplication for logic safety mechanism

In this case, the functional and safety blocks must be routed independently of one another during synthesis; otherwise, a phenomenon known as Dependent Fault Initiator (DFI) may occur. A DFI arises when a single root cause propagates across the circuit, affecting all of the redundancy components in the same way. To prevent this, during synthesis, the clock to all the redundancy components is routed independently. For instance, if the clocks to both Functional and Safety blocks were not routed separately, then a fault in the common clock will trigger a problem in both the blocks similarly causing both the outputs to fail in the same manner, thus the detect will go unnoticed. Because of this, all the possibilities of common cause effects must be eliminated during the design implementation. Due to the process, there is a slight skew that is introduced in the clock tree of the two components. This skew is one of the design traits of the ASIC flow, thus to imitate this skew in FPGA, a high-frequency clock of 200MHz was used to create a skew between the redundancy components as shown in Figure 5.3. The skew between the two clocks of redundant logic must be less than half the period of the clock; otherwise, a logic error will occur.

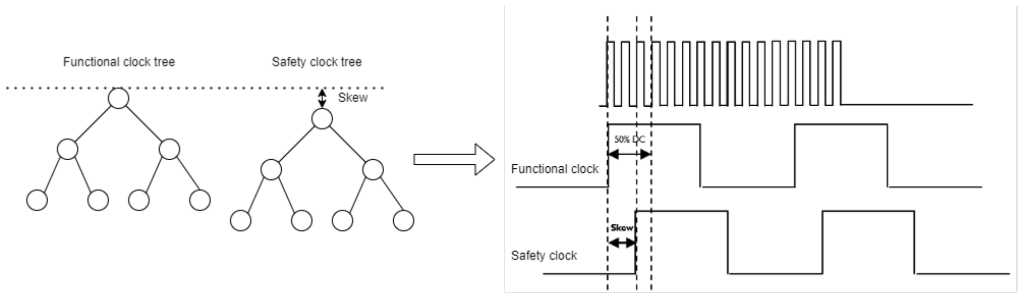


**Figure 5.3:** Skew due to DFI avoidance in ASIC flow imitated in FPGA

Once the netlist is generated, before programming it on the FPGA, it is important to build a provision that allows the fault to be injected in every node of the gate in the netlist. All these nodes must be connected with each other in a chain-like fashion. Moreover, the fault injection in any of the nodes in the fault chain must be controllable and observable. Additionally, the fault's activation and deactivation time must be programmable at runtime. The next section demonstrates the process of building the fault injection logic and how the insertion of these cells is automated using the Python framework.

## 5.3. Modifying the GL netlist

The modification of the netlist is done only once at the beginning of the campaign. To achieve this task with minimum manual work, a framework is required that recognizes the structured Verilog or SystemVerilog netlist, decomposes the various elements of the code, and then translates the desired parts. ANTLR (ANother Tool for Language Recognition) [45] parser was used for this, which acts as a generator for reading, processing, executing, or interpreting structured text SystemVerilog code. Figure 5.4 illustrates various steps involved in the modification of the netlist using ANTLR in the Python framework.

This tool employs two grammar files, **VerilogLexer.g4** and **VerilogParser.g4**, which include a series of

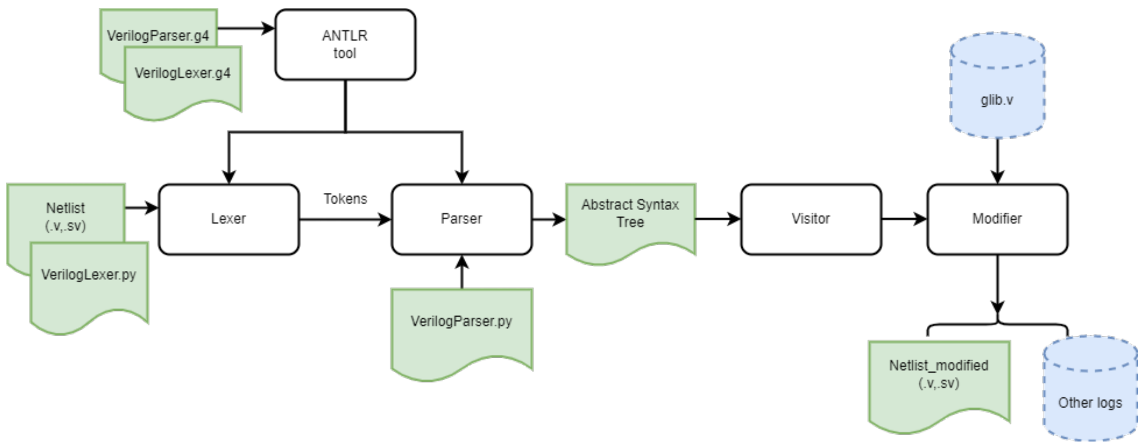


**Figure 5.4:** Block diagram demonstrating the process of translation of the SystemVerilog netlist with fault saboteurs

tokens and actions that specify how Verilog and SystemVerilog code structures should be deconstructed, and each piece of the code is parsed as a token. The **VerilogLexer.py** then uses these grammar files to create the tokens for the Verilog or SystemVerilog input file, in this case, the netlist is the input file which is tokenized. The VerilogParser.py uses these tokens to generate an Abstract Syntax Tree (AST) along with two other Python files known as Listener and Visitor which are tree-walking mechanisms in its runtime library [58]. Both use a depth-first algorithm. The primary distinction between the two is that the process of traversing the tree cannot be controlled via a listener. The routines can only be invoked when the tree's walker encounters a node of the relevant kind. Whereas, the path of the tree can be customized using the visitor; the values can be returned from each visitor function and the visitor can be stopped traversing at any point of the tree. In this work, the **Visitor** was used to traverse to each module instance definition in the AST and list all the nodes of those instances. Finally, the **Modifier** verifies the list of all the instances in the netlist using the technology standard cells database, glib.v, as shown in the figure, and then edits all the input nodes in the netlist to include the fault injectors. The fault saboteurs as shown in Figure 5.5 areinserted only at the inputs of the gate to minimize redundancy and optimize the area on the FPGA. All of the fault saboteurs are linked in series through a long shift register in a chain-like pattern via all of the gates in the netlist as illustrated in Figure 5.6. The modification of the netlist is run in an automatic fashion using a Python script. This script uses the ANTLR package which decomposes the SystemVerilog (SV) netlist into an abstract syntax tree. In this case, every instance of a gate in the SV code generates a sub-branch, with the signals acting as the leaf node. These leaf nodes are then changed with additional code to provide fault injection hooks. The resulting file of this procedure is **Netlist_modified.sv**. Along with this, various log files are generated, which include a record of all translations done to the original netlist.

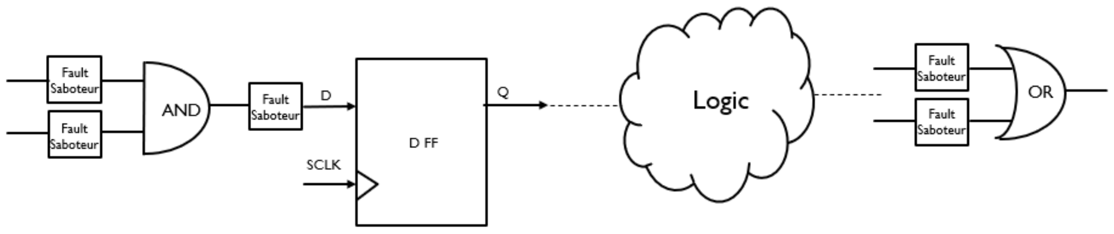


**Figure 5.5:** Adding fault saboteur at every input of the gate in the netlist

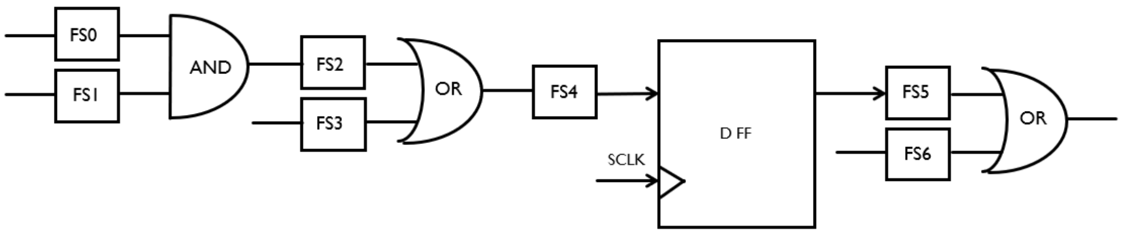


**Figure 5.6:** All fault saboteurs form a fault chain pattern

### 5.3.1. Design of Fault Saboteurs

As discussed above, the netlist generated by the synthesis tool is not ready to perform fault injection on the FPGA. To inject a fault at a particular node of the gate requires some additional control logic, which activates and deactivates the fault. These are known as Fault Saboteurs (FS), the block diagram of which is displayed in Figure 5.7.

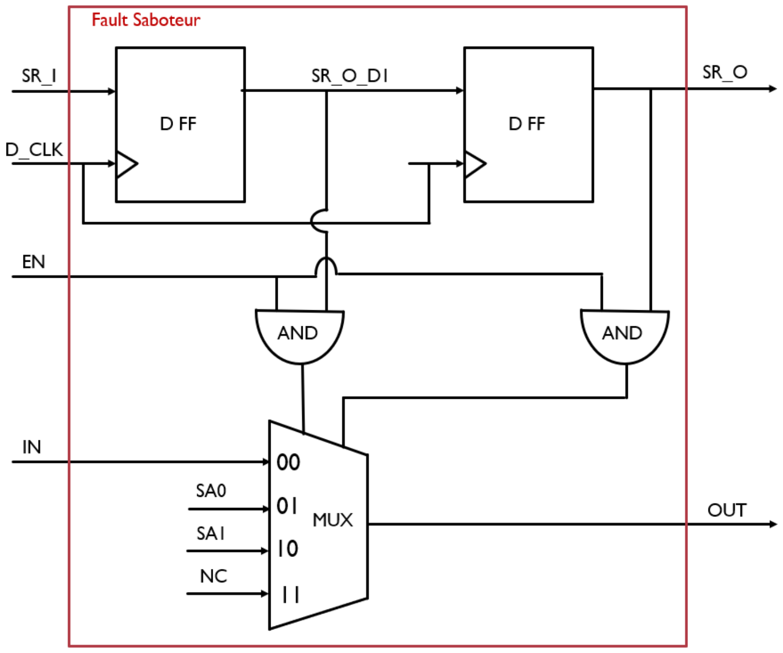


**Figure 5.7:** Block diagram of Fault Saboteur

The FS's primary function is to generate stuck-at-1 and stuck-at-0 faults that can be controlled at runtime. Furthermore, because the FS would be tied only to the input of every gate in the netlist as illustrated in Figure 5.8. As shown in the block diagram above, the FS circuit consists of 4 inputs and 2 outputs and their functionality is as below,

1. **D_CLK** is a clock signal used to shift the sequence in the FS chain to inject fault as a desired node.
2. **SR_I** is the shift register input.
3. **SR_O** is the shift register output. This signal is tied to the shift register input of the next FS.
4. **IN** is the functional input.
5. **OUT** is the functional output. This signal is the output of the multiplexer.
   When SR_O_1 and SR_O are 0,1, a stuck-at-0 fault is created and the OUT goes to zero as soon as EN goes high, otherwise the output OUT = IN.
   When SR_O_1 and SR_O are 1,0, a stuck-at-1 fault is created and the OUT goes high when EN goes high at runtime, otherwise the output OUT = IN.

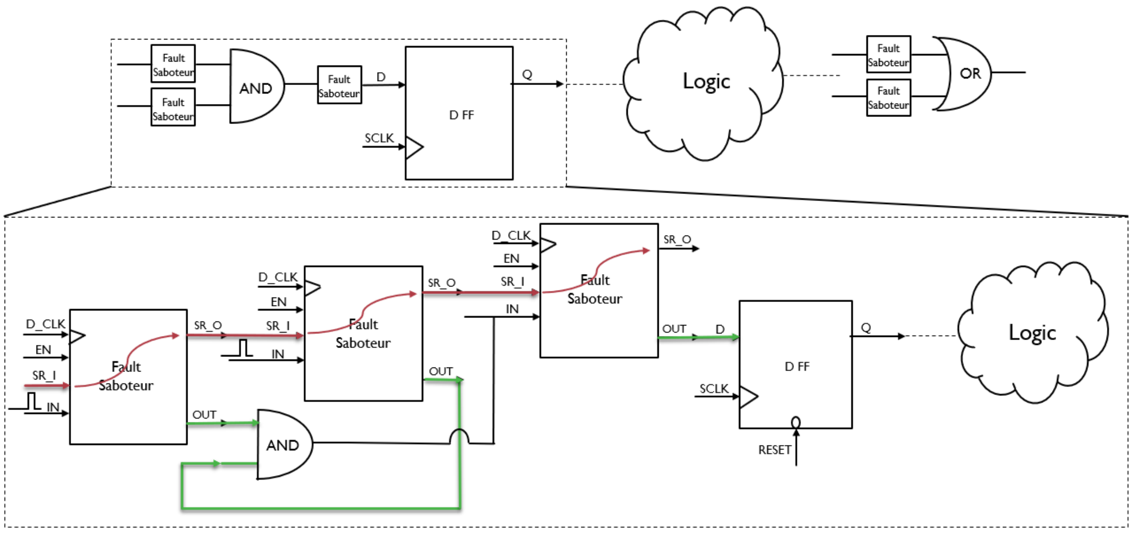


**Figure 5.8:** All fault saboteurs form a fault chain pattern

6. **EN** is a global enable signal that helps in controlling the timing of fault activation and deactivation. When this signal is held low, there is no fault injected and the functional input IN is directly passed to the functional output OUT. When this signal goes high, the corresponding SA1 or SA0 fault is activated. This signal greatly helps in activating the fault during the runtime. For instance, this signal provides flexibility to fault the fault after the startup diagnostics of the design. Additionally, it can also create a transient fault where the node can be tied to high or low only for a certain time interval.

### 5.3.2. Insertion of Fault Saboteurs

The Fault Saboteur (FS) is automatically inserted into the netlist using a Python-based script described in section 5.3. The insertion of FS is demonstrated by using a sample netlist of 1 AND gate, 2 OR gates, and a D-flipflop. The FS must be attached to every gate input without altering the functionality. This is shown in Figure 5.9.

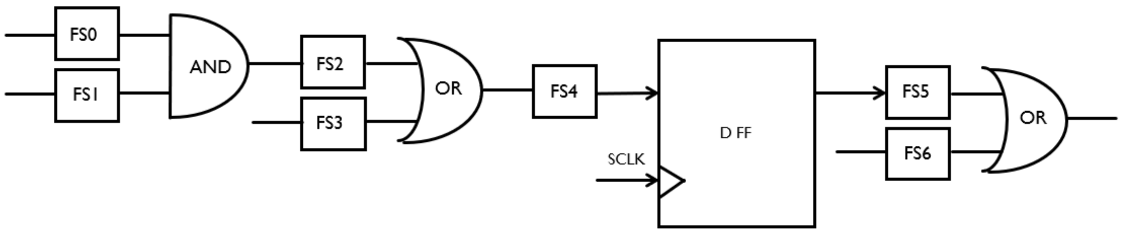


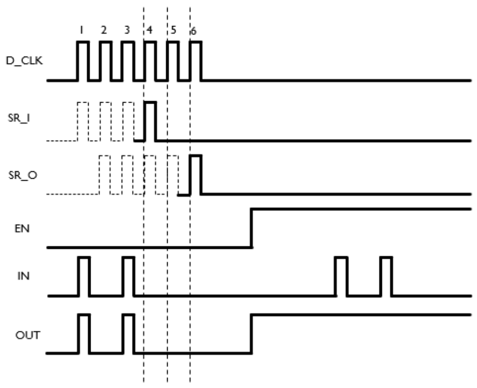
**Figure 5.9:** All fault saboteurs form a fault chain pattern

In this work, three forms of fault campaigns are tested as detailed below,

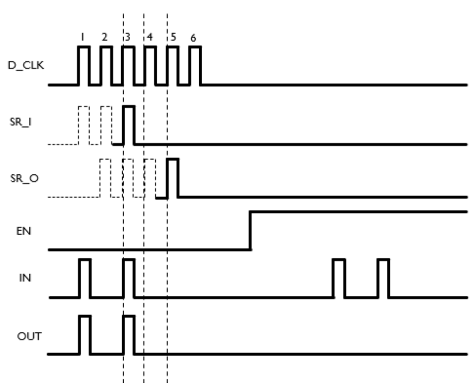
1. **Single Point Fault** occurs when just one fault is injected into the design at a time. Once enabled, this defect remains inserted until the system behavior is evaluated.
2. **Dual Point Fault** where two faults are present at the same time. In this case, both faults are activated at the same time and stay stuck until the safety sequences are executed.
3. **Transient fault** is created by inserting one fault at a time. This fault can be activated and deactivated at runtime during the execution of safety sequences.

First, consider single point fault injection at FS2. Since each FS takes two shifts to inject a fault, introducing a fault at FS2 will require six clock cycles. The pattern to be shifted for SA1 and SA0 is

000001 and 000010 respectively. This is demonstrated using the waveforms shown in Figure 5.10a and Figure 5.10b.

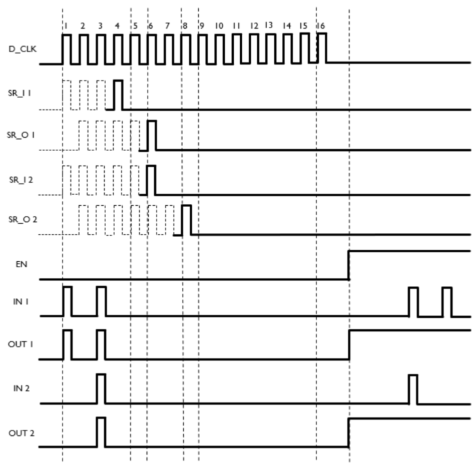


**(a)** Single point stuck-at-1 fault at FS2

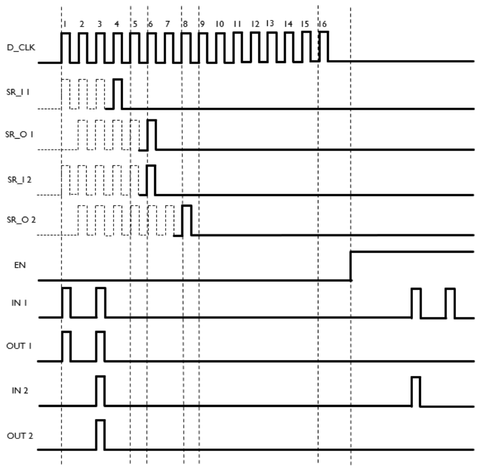


**(b)** Single point stuck-at-0 fault at FS2

Second, the method of fault injection in dual point faults differs from that in SPF. When two faults are to be produced, the fault saboteur site cannot determine the number of clock cycles. In this case, the total number of clock cycles is equal to the number of fault saboteurs in the netlist. Consider, for example, introducing faults at FS2 and FS4. Figure 5.11a and Figure 5.11b depicts Stuck-at-1 and Stuck-at-0 respectively.

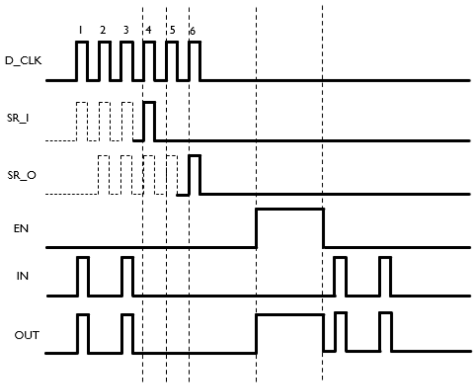


**(a)** Dual point stuck-at-1 fault at FS2 and FS3

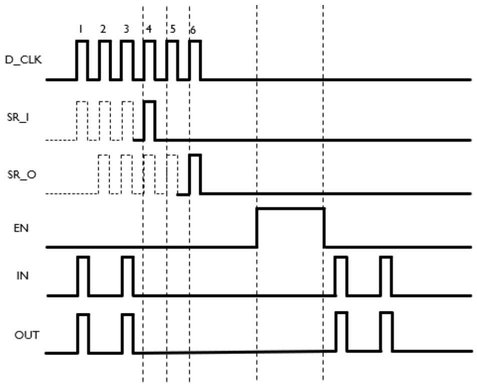


**(b)** Dual point stuck-at-0 fault at FS2 and FS3

Lastly, the transient fault is created in the same manner as the SPF. However, in this case, the fault activation and deactivation may be controlled at runtime. This is illustrated in Figure 5.12a and Figure 5.12b.



**(a)** Transient fault at FS2 where output is held high momentarily



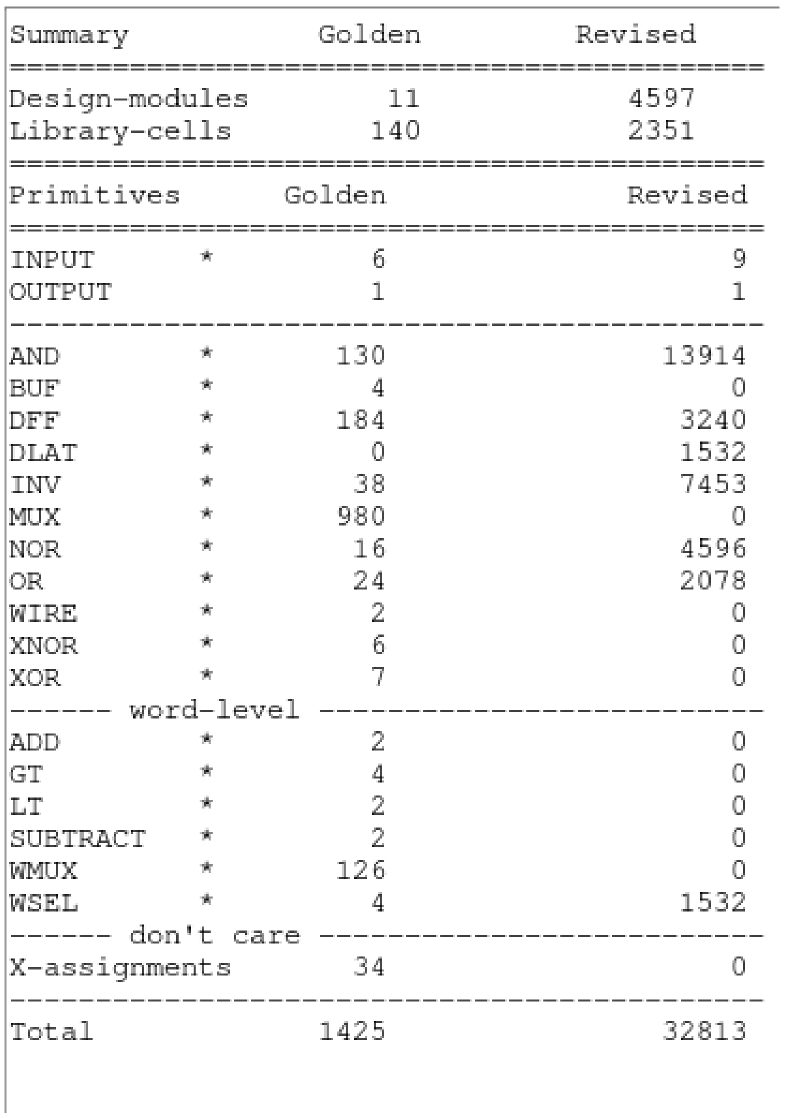
**(b)** Transient fault at FS2 where output is held low momentarily

## 5.4. Logic Equivalence Check

The final step in design preprocessing is the Logic Equivalence Check between the Design Under Test and the revised netlist with fault injectors. This activity is critical for concluding that the modifications do not affect the functionality of the netlist. To perform this activity Conformal tool by Cadence was employed [14].

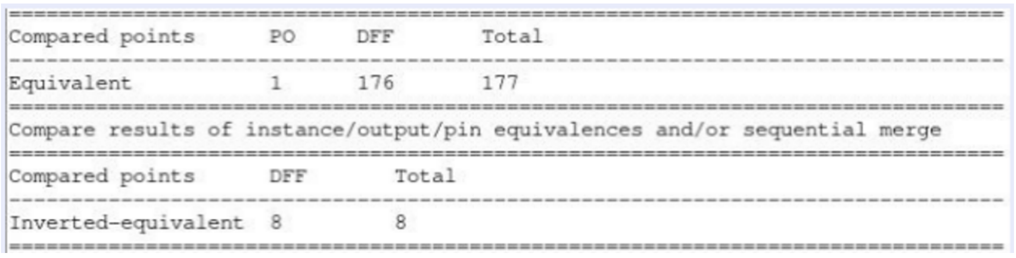
The technique requires two independent revisions of a design with the same expected functionality. Conformal then compiles the RTL descriptions of these designs, creating an internal representation. The tool's core functionality revolves around meticulously comparing these internal representations to ensure functional equivalence. It examines combinational and sequential logic, as well as interconnections, seeking any differences.

To begin with, the Design Under Test RTL design in Verilog is supplied as the golden design. This is then elaborated using the technology library file same as the file used for GLS. The modified netlist is provided as the revised design. To elaborate the modified netlist, the Verilog definitions of the technology library as given. Once, both the golden and revised designs are elaborated. The constraint is applied to the revised netlist to bypass the fault injection mechanism. That is, the clock used to activate the fault saboteur $D\_CLK$ is set to zero. In addition, the global enable signal $EN$, which activates the fault, is set to zero. This signifies that the fault injection logic in the modified netlist has been disabled. The tool is then set to compare the DUT and the modified netlist. Figure 5.13 illustrates the equivalency report obtained after elaborating on both RTL design and the modified netlist at the gate level. It displays the number of gates present in each version of the design. Now, both designs are compared with the constraint explained above. Figure 5.14 shows the equivalence check output by conformal. It can be seen that there are no non-equivalent points present between the two designs indicating that under the condition of global fault enable $EN$ and fault saboteur clock $D_CLK$ tied low, both the designs are equivalent. This activity concludes that the design has been successfully transformed to map onto the FPGA.



```
Summary              Golden          Revised
=============================================
Design-modules          11            4597
Library-cells          140            2351
=============================================
Primitives       Golden                Revised
=============================================
INPUT       *         6                     9
OUTPUT                1                     1
---------------------------------------------
AND         *       130                 13914
BUF         *         4                     0
DFF         *       184                  3240
DLAT        *         0                  1532
INV         *        38                  7453
MUX         *       980                     0
NOR         *        16                  4596
OR          *        24                  2078
WIRE        *         2                     0
XNOR        *         6                     0
XOR         *         7                     0
------ word-level -------------------------
ADD         *         2                     0
GT          *         4                     0
LT          *         2                     0
SUBTRACT    *         2                     0
WMUX        *       126                     0
WSEL        *         4                  1532
------ don't care -------------------------
X-assignments        34                     0
---------------------------------------------
Total              1425                 32813
```

**Figure 5.13:** Logic Equivalence Check report for design vs modified netlist with fault saboteurs



```
================================================================================
Compared points      PO     DFF       Total
--------------------------------------------------------------------------------
Equivalent           1      176       177
================================================================================
Compare results of instance/output/pin equivalences and/or sequential merge
================================================================================
Compared points      DFF       Total
--------------------------------------------------------------------------------
Inverted-equivalent  8         8
================================================================================
```

**Figure 5.14:** Logic Equivalence Check output for design vs modified netlist with fault saboteurs

# 6 Design and Verification

The fundamental goal of this project is to create a fast and effective platform for fault emulation on the FPGA. As introduced in chapter 4, the main task of this system is to acquire campaign information from the host PC, categorize the various data, run the fault campaign, execute the functional and safety sequences, and then provide the DUT output back to the PC for fault classification. Based on the different tasks, four interfaces are realized as illustrated below,

1. The **Communication interface** is responsible for serially transmitting the campaign data from the host PC to the FPGA.
2. The **Fault Interface** controls the injection of a fault (Stuck-at-1, Stuck-at-0) at a time for different type of campaigns (SPF, DPF, TF).
3. The **Functional Interface** is used to execute the functional and safety pattern on the Design Under Test.
4. The **Analog Interface** enables communication between Analog and Digital components. However, because the DUT architecture is entirely digital, this interface was not evaluated in this study. However, the fault emulation system developed is capable of supporting this interface.

In this work, two designs of fault emulation systems have been investigated. The first architecture was developed with a minimum number of hardware components required to perform the task discussed above. This architecture was heavy on software programming which caused the major issues in the latency due to the Operating System (OS) and high-level Python libraries like Pyserial and Pyftdi. To minimize the software overhead, a Microcode-based architecture was built. This platform is a good balance between the software and hardware implementation. This chapter covers the two designs and highlights the advantages as well as the disadvantages. A brief verification of the system is also discussed.

## 6.1. Purely Programmable-based Architecture

To begin with, a simple architecture was drawn as shown in Figure 6.1. This consists of three main interfaces, namely the communication interface, the fault interface, and the functional interface.

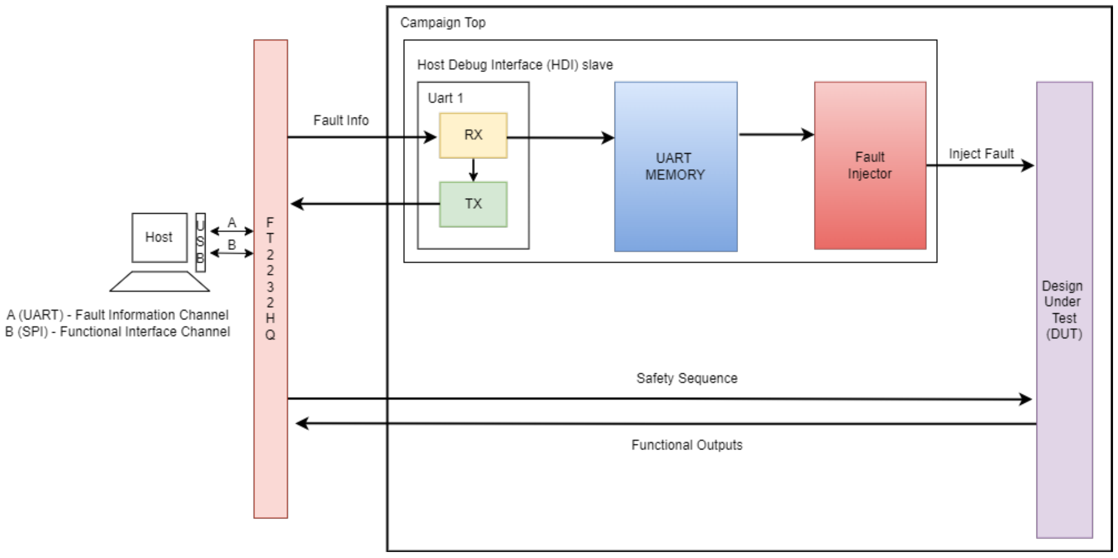


**Figure 6.1:** The initial architecture of HDI slave

The communication interface is used to send fault data from the PC to the FPGA. For this purpose, a high-speed FT2232HQ FTDI chip is employed. This is a popular USB-to-serial bridge controller manufactured by Future Technology Devices International (FTDI). It features two independent UART/FIFO channels and is commonly used for interfacing USB with various serial communication protocols such as UART, SPI, I2C, and JTAG. The FT2232HQ is widely employed in embedded systems, programming/debugging tools, and other applications requiring USB connectivity for serial communication. In this architecture, one of the UART channels is used as a fault information channel. The Host Debug Interface (HDI) master is a piece of software on the PC that accepts campaign-related information from the user and sends it to the FPGA through this channel.

Once fault information from the PC is received through the UART, it is processed and made accessible to the fault injector. The fault interface's task is to govern the injection of a fault into the DUT. This interface precisely controls the fault's activation and deactivation during the runtime of the safety routines. For example, in the case of a single-point or dual-point fault, the fault remains active at the specified location throughout the execution period of the functional and safety sequences. However, for transient faults, the duration of the fault's active time can be controlled using the fault interface.

Finally, once a fault has been introduced into the system, the functional interface facilitates system evaluation by performing the safety sequences. Section 7.1 describes the safety sequence tested in this case. The functional interface, which is the FT2232HQ chip's second serial communication channel, is developed utilizing SPI master written in Python.

### 6.1.1. Software Implementation

The Host Debug Interface (HDI) master is a piece of software that is mainly responsible for two tasks. Firstly, collect the campaign configuration data from the user and transmit it serially through UART to the FPGA using channel A in the block diagram. The communication bus for this architecture with fault information is shown in Figure 6.2. As demonstrated in Figure 6.3, the TOC field contains information about the type of campaign. The next two fields include information about the fault site. This is done using two 24-bit variables. Only fault site 1 is relevant in the event of Single-Point Fault (SPF) and Transient Fault (TF), whereas fault site 2 is kept at zero. Fault site 2 is used for Dual-Point Fault (DPF). The following two 24-bit values represent the fault's activation and deactivation times. The last two values provide the assertion and de-assertion times of the Design Under Test reset.

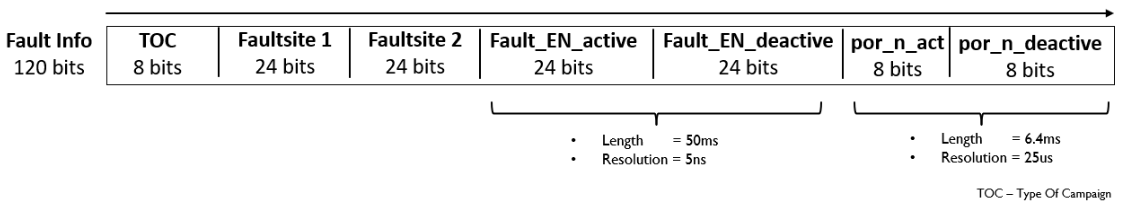


**Figure 6.2:** Communication bus for fault campaign

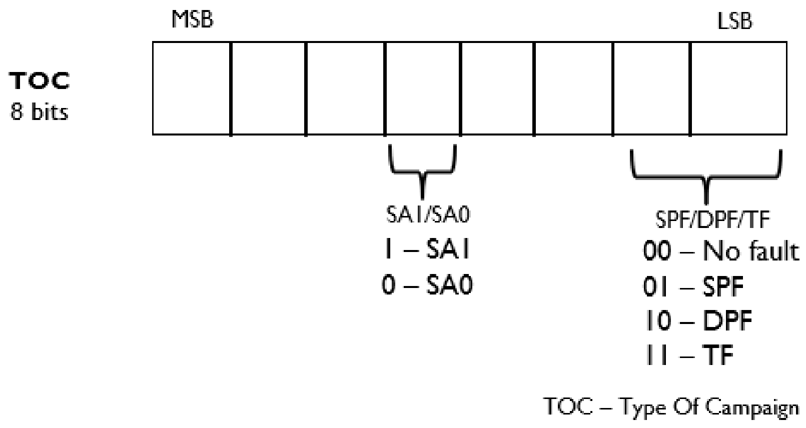


**Figure 6.3:** Type of campaign field of the communication bus

The second task of the HDI master is to execute the functional and safety sequences on the Design Under Test through the SPI protocol using channel B.

During the implementation of these two channels, it was realized that this architecture adds a lot of software overhead which greatly affects the performance of the fault campaign. Channel A employing UART was implemented using the Python library pyserial [38]. This adds software overhead primarily due to the way serial communication works and how the operating system manages it. When the data is sent or received over a serial port, it is usually done byte by byte. PySerial often buffers data to ensure a consistent and reliable flow of bytes, which introduces overhead. Moreover, the operating system handles the low-level communication with the serial port hardware, which introduces some overhead. This includes context switching between the application and the OS kernel. Additionally, serial communication usually involves sending binary data, but Python operates with text-based strings. Thus, encoding and decoding operations also introduce some overhead.

Similarly, the second channel implemented using the SPI controller library (spi.controller) in PyFtdi [7] provides a Python interface for interacting with Serial Peripheral Interface (SPI) devices using FTDI USB-to-SPI bridges. While PyFtdi is designed to be efficient and provide a high-level interface for SPI communication, it does introduce some software overhead. Firstly, PyFtdi communicates with FTDI USB devices over the USB bus. USB communication inherently involves software overhead because the data must be serialized, sent over the USB bus, and then deserialized on the other end. This involves interaction with USB device drivers and the operating system's USB stack. Secondly, being a Python library, PyFtdi operates within the context of the Python interpreter. This introduces some overhead due to Python's dynamic typing, memory management, and interpretation of Python code. Additionally, PyFtdi uses internal buffers to optimize the data transfer. These buffers introduce a small amount of software overhead to manage and maintain them.

As a result of numerous factors, the two interfaces contribute a significant amount of overhead. The communication channel adds a $12ms$ overhead for each fault, while the functional interface adds a $7ms$ overhead for every safety sequence. This demonstrates that when the number of safety sequences rises, the time required to execute the safety pattern increases significantly, adding to the overall campaign latency.

Thus, taking into consideration the software overhead for both channels, it was determined that this architecture needed to be improved in order to resolve the latency problem to accomplish the project's performance objective of execution time for the campaign of less than 3 hours.

The fault emulation findings and performance data for this architecture are shown in section 7.2. The limitations of this design were identified and the next section shows the enhanced version of the emulation system.

### 6.1.2. Advantages

1. Despite the fact that this design adds a lot of software overhead, it is versatile in terms of fault setup. Here, a single fault with SA1 or SA0 fault can be specified individually for SPF, DPF, or TF.
2. Second, this approach allows for greater flexibility in executing safety sequences. This enables the execution of any form of sequence, such as conditional or branching sequences.
3. Thirdly, this design is easily adaptable to any DUT. If the DUT employed serial protocol other than SPI as used in this case. It is simple to implement using the PyFtdi serial library.
4. Additionally, in situations where the DUT consists of ADCs with long conversion times, the software overhead is relatively substantial. In this situation, the real-time required to complete the operation exceeds the software overhead, hence the software overhead is no longer a constraint.

### 6.1.3. Limitations

1. The major limitation is the software overhead which adds huge latency to the execution time of the fault campaign. This overhead is unavoidable in serial communication libraries like PySerial and PyFtdi due to the fundamental requirements of data serialization, buffering, operating system interaction, and high-level language execution.

## 6.2. Architecture 2

The prior design failed to meet the fault campaign's target execution time. The main challenge was the software overhead in the communication and functional interfaces. The latency was mainly due to the operating system and the usage of high-level Python libraries pyserial and pyftdi.spi for UART and SPI communication respectively. The two interfaces contributed an estimated $54ms$ overhead per failure, which is a significant amount for running a 1M campaign.

In this architecture, the overhead problem was addressed in two ways. First, the UART protocol was exploited to determine the maximum number of bytes that can be transferred in a single packet of UART transactions. Figure 6.4 displays the graph of the time taken to transfer 15M bytes of data through UART in various chunk sizes per transaction. It can be observed that as the number of bytes per transaction increased, the time taken reduced significantly. However, there is a limit to the maximum amount of bytes that may be sent without any data packets being lost. This limit is determined by several factors, including the USB protocol itself and the capabilities of the FTDI chip and its associated drivers. Firstly, the USB is a packet-based protocol, and data is transmitted in packets or frames. The maximum packet size for a USB endpoint is determined by the USB specification (e.g., USB 2.0 or USB 3.0). In this work, USB 2.0 was exercised. Additionally, the FTDI chip itself has limitations on the maximum payload it can handle in a single USB transaction. Moreover, it was noticed that above 8K bytes per transaction, the reduction in the time taken on an average per byte was marginal. Table 6.1 presents the time taken for data transfer using UART for different chuck sizes. From this experiment, it was observed that data transmissions exceeding 9000 bytes exploded the transmit and receive buffers, resulting in communication failure. This indicated that the maximum limit had been reached. Because the difference in average time per byte between 7500 and 9000 bytes per transaction was minimal, and to avoid data loss in very high packet sizes, 7500 bytes per transaction was determined to be the optimum bytes per transaction to safely transfer the data using UART. This led to the idea of sending information for multiple faults at a time.

The second issue was the overhead in the SPI communication when the safety routines were being executed. This overhead could be minimized in two ways. One approach was to implement a high-speed

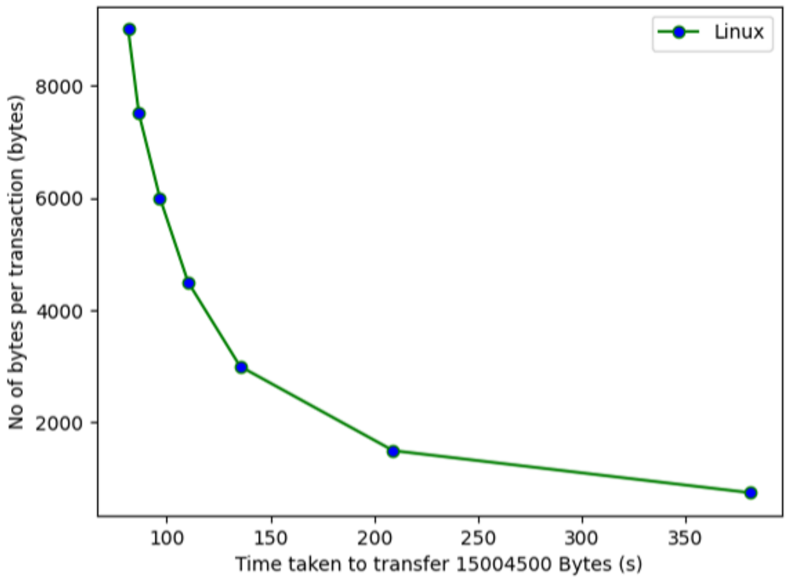


**Figure 6.4:** Data transfer through UART using Python in Linux operating system

microcontroller on the FPGA alongside the DUT, that can run the safety routines with minimum overhead between the consecutive sequences. This prompted the development of low-level device drivers in C-programming for performing the functional interface. The second method was to implement a microcode-based architecture that can decode the safety sequences and execute them. The latter was chosen considering the software and hardware implementation complexity.

| **Bytes per transaction** | **No of transactions to transfer 15M bytes** | **Time taken (s)** | **Average time per byte ($\mu s$)** |
|---|---|---|---|
| 750 | 20006 | 382 | 25.4 |
| 1500 | 10003 | 209 | 13.9 |
| 3000 | 5002 | 135 | 9.02 |
| 4500 | 3335 | 110 | 7.35 |
| 6000 | 2500 | 96 | 6.44 |
| 7500 | 2000 | 86 | 5.77 |
| 9000 | 1430 | 81 | 5.43 |

**Table 6.1:** Time taken to transfer 15M bytes of data through UART in varying chunk size per transaction

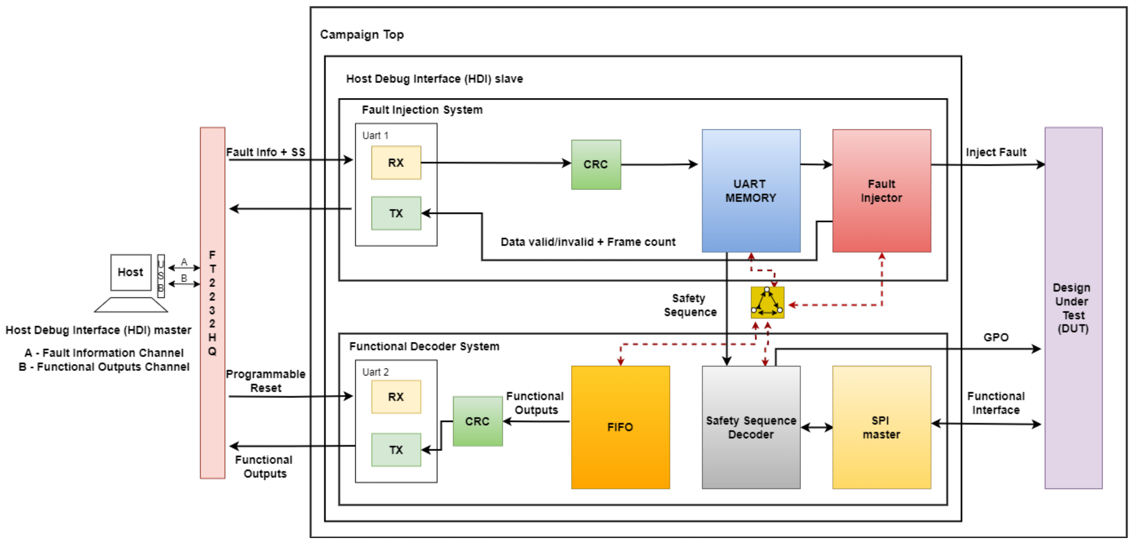


**Figure 6.5:** Architecture 2 of the fault emulation system

The block diagram of the final architecture is presented in Figure 6.5. The design is divided into two major components: the Fault Injection System (FIS) and the Functional Decoder System (FDS). As seen in the diagram, they comprise the Host Debug Interface (HDI) slave. The FIS is primarily responsible for injecting the fault into the DUT, whereas the FDS is in charge of executing the safety procedures. In this architecture, the design of the two interfaces that produced the significant overhead is reimagined. Instead of using UART and SPI for communication and functional interfaces, the HDI master uses two UART channels, one for conveying fault injection and the other for DUT functional outputs. In the next section, the function of each block is explained.

### 6.2.1. Building blocks of the architecture

The campaign manager on the FPGA denoted as the Host Debug Interface (HDI) slave in the block diagram consists of seven major sub-blocks, namely the two UART modules, one for fault information and the other for functional outputs, second, the UART Memory to store the data received from the HDI master, Fault Injector module to create faults in the Design Under Test, Safety Sequence Decoder to decode the three types of safety commands discussed in the previous section and the SPI master to communicate with the DUT.

To begin with, the UART1 receiver takes the incoming data from the HDI master on the PC side and validates it using a 16-bit CRC polynomial CCITT. A cyclic redundancy check (CRC) is a program that detects errors in digital networks and storage devices. Data blocks entering these systems are assigned a brief check value depending on the remainder of a polynomial division of their contents. When the data is retrieved, the computation is performed again, and if the check values do not match, remedial action against data manipulation can be taken [15]. In this work, the CRC-16 CCITT with polynomial 0x1021 is used. Due to the large amount of bytes in the transaction, the CRC module is added to ensure that no data bytes are lost during transmission. Only once the CRC block validates that the correct information has been received is it written to memory and made available to other blocks.

To communicate between distinct blocks in a regulated manner, a finite state machine is implemented. This is illustrated in Figure 6.6 with a bubble diagram. The machine enters the **FRAME_START** state only when the two variables **p_uart_data_rcvd** and **data_valid** are asserted, as shown in the figure. The variable **p_uart_data_rcvd** indicates that the proper number of bytes have been received, and **data_valid** ensures that the information received is accurate.

Moving forward, after the data has been validated by the CRC, it is written to the UART Memory and categorized into several fields. The next block fault injector takes the fault information, interprets the type of campaign and faults, and then injects one fault at a time. The start of the fault injector activity is shown in the bubble diagram by asserting the variable **fault_injection_start**. The injector

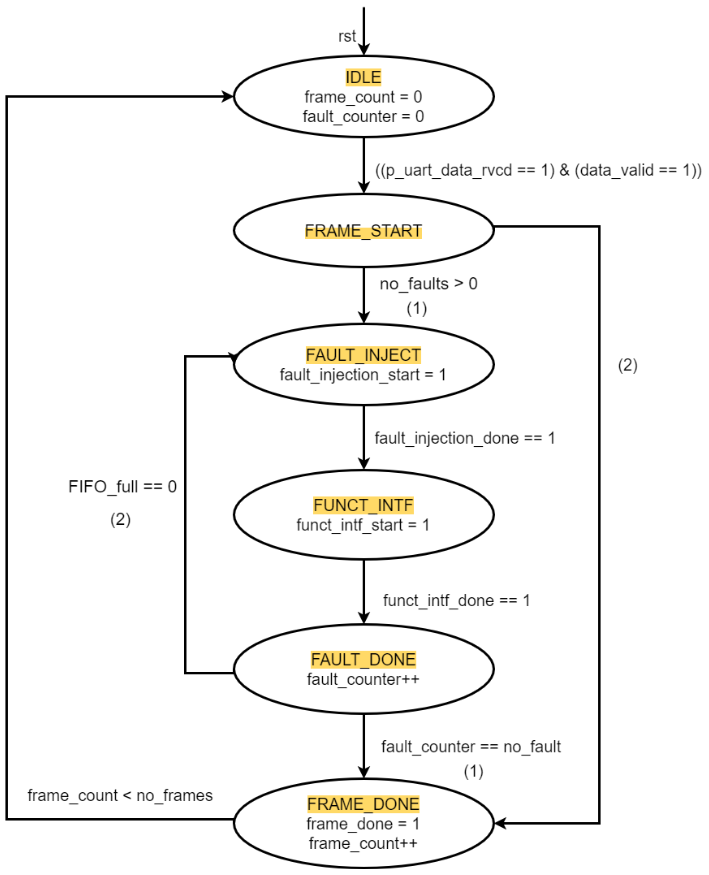


**Figure 6.6:** Finite state machine illustrating the steps in executing the fault campaign

sets the variable **fault_injection_done** to high after the fault is produced at the specified location. The **FAULT_INJECT** state is now complete.

Next comes the **FUNCT_INTF**, where the functional and safety sequences are executed. The UART memory supplies the safety sequence and the number of command information to the Safety Sequence decoder block. This module decodes the sequences and prepares the functional interface to be carried out by the SPI master. The start of execution of sequences by the SPI master is indicated by the variable **funct_intf_start**. Once all the commands are run, the end of this stage is signaled by setting the flag **funct_intf_done** high. Additionally, the output of the Design Under Test is recorded by the Safety Sequence decoder and put onto the FIFO to be transmitted back to the HDI master for fault classification through the UART2 transmitter.

This completes a full cycle of evaluating the system for a single fault. In the next state **FAULT_DONE**, the fault counter is incremented, and if there is sufficient space in the FIFO to accommodate the functional output, the injection of the next fault is initiated. The cycle continues until all the faults in the frame of UART are examined. The total number of faults per transaction is determined from the Campaign Preamble variable $n_F$. Once the frame is complete, the system goes to the final state **FRAME_DONE**. The frame counter is incremented and the completion of the frame is indicated to the HDI master along with the frame counter through the UART1 transmitter. Once, the HDI master receives this packet, the next transaction is initiated. Similarly, all the transactions are executed and the final execution time is recorded for the performance evaluation of this project.

In addition, if the FSM becomes stuck in any of the states and enters the deadlock state, the HDI master provides a timeout mechanism. Depending on the requirements of the Design Under Test, a specified timeout is configured. If the frame completion signal is not received within this time limit, a reset signal can be programmed using the UART2 receiver to reset the HDI slave and restart the campaign.

### 6.2.2. Software Implementation

The drawbacks of the prior architectural software implementation were identified, and this architecture delivers an improved version that offers promising performance in campaign execution time. To begin, the communication bus in this architecture has been optimized, as seen in Figure 6.7. As previously stated, the ideal amount of bytes per UART transaction is 7500. Table 6.2 shows configurations of the different parameters involved in the communication bus that has been optimized based on the optimum bytes in the UART transaction.

| Symbol | Value | Unit | Description |
|---|---|---|---|
| $n_{SC}$ | 63 | | Number of safety commands |
| $n_F$ | 1023 | | Number of faults in a single transaction of UART |
| $C_{pre}$ | 16 | bit | $n_F + n_{SC}$<br>Campaign Preamble |
| $F_{info}$ | 72 | bit | Fault Information<br>Type of campaign, fault activation, and deactivation time |
| $F_{sites}$ | $24 \times 1023$<br>24552 | bit | Fault sites<br>Relative position of fault in the XCELL chain |
| $S_{SS}$ | $2^{15}$<br>32768 | bit | Safety Sequences<br>Numerous safety sequences encoded one after the other |
| CRC | 16 | bit | CRC16-CCITT |
| $T_F$ | 57440<br>7180 | bit<br>bytes | Total number of bits in the communication bus |

**Table 6.2:** Configuration of parameters present in the communication bus

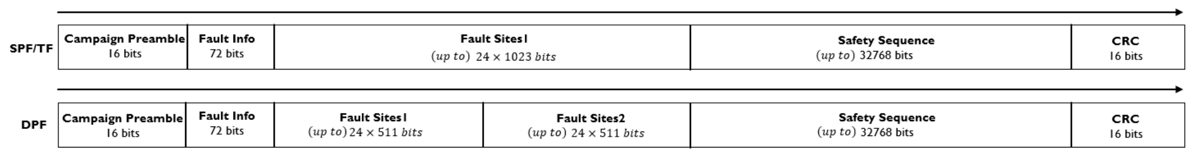

SPF/TF
Campaign Preamble 16 bits
Fault Info 72 bits
Fault Sites1 $(up\ to)\ 24 \times 1023\ bits$
Safety Sequence $(up\ to)$ 32768 bits
CRC 16 bits
DPF
Campaign Preamble 16 bits
Fault Info 72 bits
Fault Sites1 $(up\ to)\ 24 \times 511\ bits$
Fault Sites2 $(up\ to)\ 24 \times 511\ bits$
Safety Sequence $(up\ to)$ 32768 bits
CRC 16 bits

**Figure 6.7:** Communication bus for architecture 2

The various parts of the communication bus are explained below,

1. **Campaign Preamble** is divided into two parts: the number of safety commands in present the long Safety Sequence and the number of faults to be tested in a single frame of UART. This is displayed in Figure 6.8.
2. **Fault Information** describes the type of campaign, the time of fault activation and deactivation, and the time of DUT reset assertion and de-assertion. This experiment evaluates three types of campaigns: single-point fault, dual-point fault, and transient fault campaigns. In addition, two types of failures are tested: stuck-at-1 and stuck-at-0. This information is applicable to all the faults investigated in a single transaction.
3. **Fault sites** include information about the location where the fault must be established. SPF and DPF encode this field differently. The site information for SPF provides relative shifts of the position, whereas the exact fault location is supplied for DPF. For instance, consider an example where the length of the fault chain is 6. The number of shifts and clock cycles required to inject the fault for SPF, DPF and TF are illustrated in Figure 6.9, 6.10 and 6.11 respectively. The time required to inject the faults is optimized to 2 clock cycles per fault, as seen in Figure 6.9. As a result, the injection speed is directly correlated to the frequency of the clock. The experiment used a 200MHz clock because that was the maximum testable frequency available for the FPGA used in this research. Furthermore, when the design netlist expands and the fault chain lengthens, the time required to inject the fault does not change. As a result, the limit for optimizing the fault interface has been reached. A comparable method, however, cannot be implemented for DPF. Because the two faults in the chain are random, the entire chain must be scanned to inject the faults. As a result, the length of the chain influences DPF injection time.

4. **Safety Sequences** is a series of commands designed to assist in system evaluation for safety. These sequences enable the testing of the safety measures implemented in the Design for safety-related applications. Three commands were developed to assess a design: Payload, Wait, and GPO. They are built in a CISC style, with the LSB 2 bits determining the type of command (0 - Payload, 1 - Wait, and 2 - GPO). The Payload command carries read/write data to be serially transmitted to the Design. The structure of the Payload command is depicted in 6.12. The LSB 2 bits are utilized to recognize this command, the next 6 bits identify the number of data bytes in the Payload, and the remaining $2^6$ bits decide whether each byte in the Payload data is a read or write type. After that, the data bytes are attached. The length of the BCD bits and OHE bits is constant, while the length of the Payload data bytes is variable based on the BCD value. In this work, this instruction is used to execute a register file read/write; the read type of data byte is indicated with a place value of 1 in the relevant OHE bit, and the write type of data byte is indicated with a place value of 0. Wait is the second type of command. This instruction can be used when the DUT has to execute certain operations and the output is only accessible after a specific time interval. For example, while reading an ADC's output, a Payload instruction can be issued to commence the ADC operation, followed by a Wait command to establish a delay that allows the ADC to conduct the operations. After a sufficient amount of time has passed, another Payload instruction may be issued to read the ADC outputs. Furthermore, when a specific GPO output is asserted, the third kind of command, GPO, may be utilized to begin carrying out a particular action in the DUT.

   All the data required by the Host Debug Interface (HDI) slave for executing the fault campaign is collected from the user using a Python-based code, packaged in the form of a communication bus as displayed in Figure 6.7 and sent to the HDI slave using channel A through UART indicated as the Fault Information Channel in Figure 6.5.

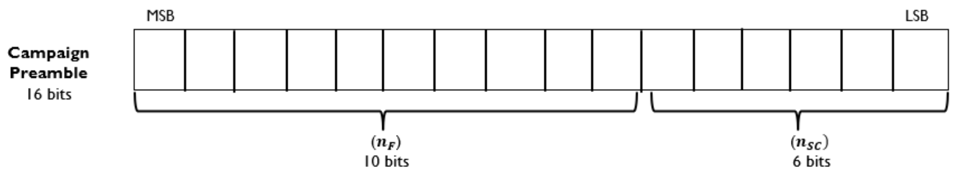


**Figure 6.8:** Campaign preamble in the communication bus

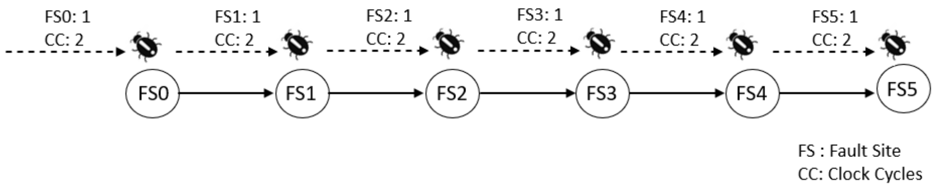


**Figure 6.9:** Example for injecting single-point fault in the fault chain

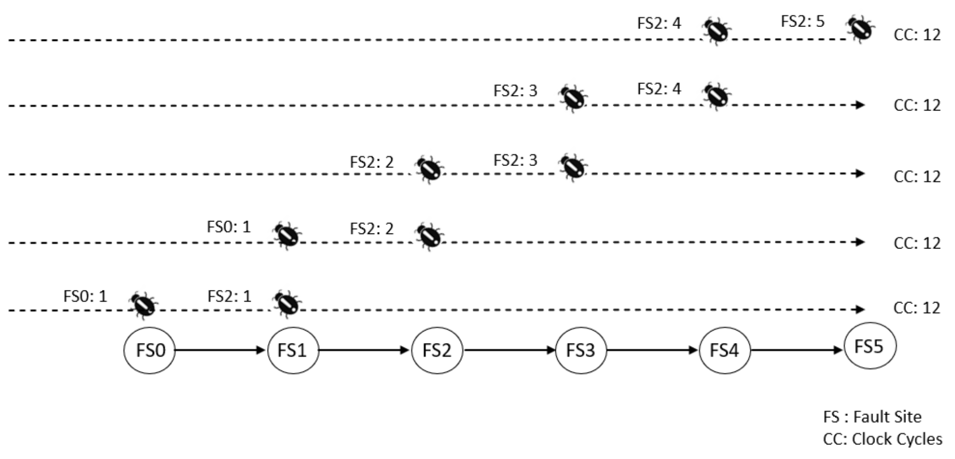


**Figure 6.10:** Example for injecting dual-point fault in the fault chain

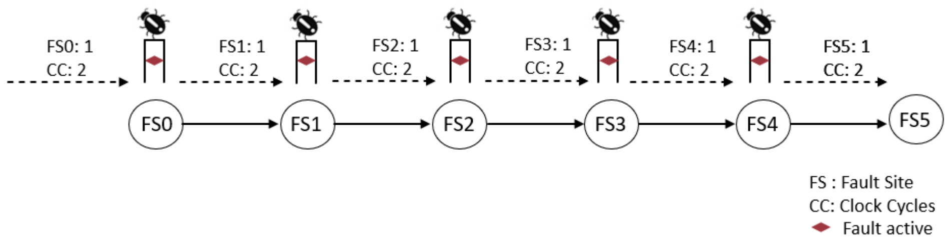


**Figure 6.11:** Example for injecting transient fault in the fault chain

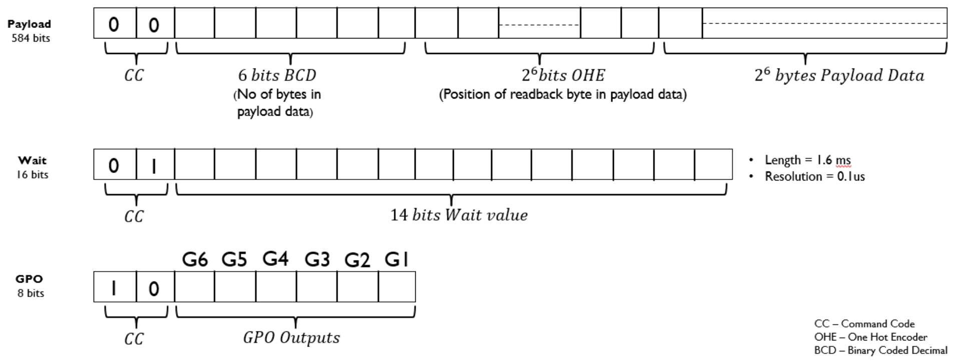


**Figure 6.12:** Safety Commands: Payload, Wait and General purpose output (GPO)

### 6.2.3. Advantages

1. All three interfaces of the campaign emulation system are designed to have minimum overhead.
2. The HDI slave and master architectures are universal enough to test majority of the designs. The only thing that needs to be changed is the functional interface master. For example, in this work, the Design Under Test includes the SPI protocol, hence an SPI master is constructed on the HDI slave side. If the Design Under Test changes and contains a different serial protocol, such as I2C, the SPI master must be replaced by an I2C master, while the rest of the HDI slave components remain untouched.

### 6.2.4. Limitations

1. Currently only three safety sequences are developed. If the Design Under Test requires additional routines to be tested. This requires a change in the CISC architecture of the safety commands.
2. The Safety Sequences are pre-determined and sent through the communication. Thus, the execution of functional routines that involve condition execution or branching is not supported by this architecture.

## 6.3. Verification methodology

Functional verification is a critical step in the design process, ensuring that digital systems operate as intended and meet their specifications, which is essential for the reliability and performance of electronic devices and integrated circuits. Traditionally, a thorough verification process is carried out, as seen in Figure 6.13. It consists of a left "V" representing the design phase and a right "V" representing the verification phase. In the design phase, high-level requirements are defined and refined into detailed specifications, while in the verification phase, tests and checks are performed to validate that the design meets those requirements.

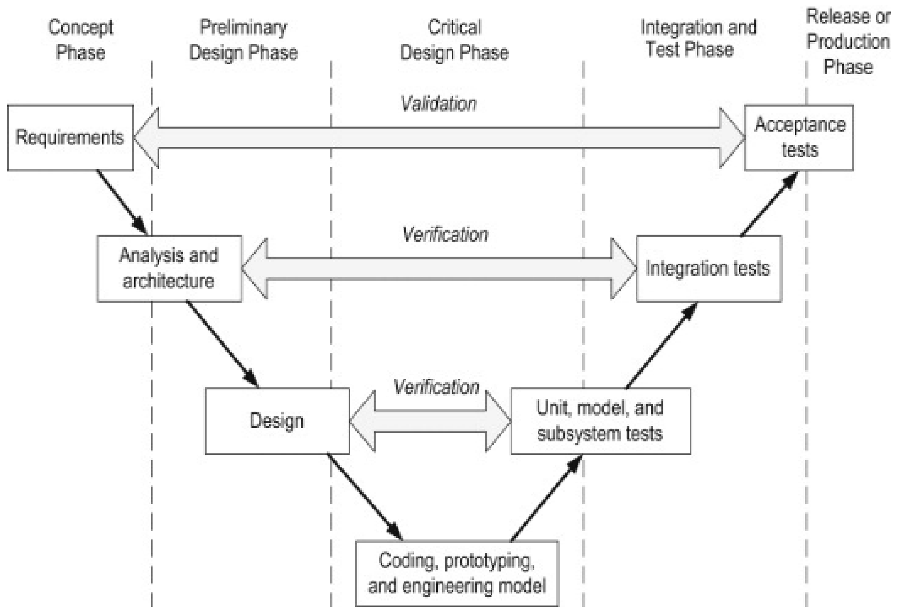

Concept Phase
Preliminary Design Phase
Critical Design Phase
Integration and Test Phase
Release or Production Phase
Validation
Requirements
Acceptance tests
Verification
Analysis and architecture
Integration tests
Verification
Design
Unit, model, and subsystem tests
Coding, prototyping, and engineering model

**Figure 6.13:** V-model shows the process of product development at system level [23]

Every design has a verification IP designed for it. This process typically follows the steps below,

1. First, the design specifications are defined and creation of a verification plan is outlined. This includes functional requirements, performance metrics, and corner cases.
2. Testbench environment using SystemVerilog and UVM is created. These test benches generate test vectors, monitor the DUT's behavior, and check the results for correctness. Then, the simulations are performed using tools like ModelSim or Cadence Incisive.
3. Additionally, assertions are embedded in the testbench to capture design constraints and properties that must hold true.
4. The coverage metrics are analyzed to determine how thoroughly the design has been tested.
5. Formal verification tools are used to perform exhaustive analysis and prove properties about the design.
6. Lastly, once all verification objectives are met and the design is deemed correct, it can be signed off for production.

However, due to the limited time available to complete this project. The system was validated utilizing directed testing during simulations, and the inspection approach was followed to validate the system on FPGA. A brief verification plan for testing the fault emulation system was drawn keeping all the critical functionality points in mind. The system was verified using the following test case,

1. Send correct campaign data through the communication interface and observe the behavior.

Based on the test case, some of the critical points for verification of the system are listed below,

1. Uart1 receiver signals the reception on data from the HDI master only when the correct number of bytes are received.
2. If the data received is incorrect, verify that the transaction is dropped and indicated to the HDI master via Uart1 transmitter along with the frame count.
3. Verify that the fault injector block injects the fault at the right location in the DUT.
4. Verify the enable of fault EN is activated and deactivated as per the designed resolution.
5. Verify the power on reset of the DUT ($por\_n$) is asserted and deasserted for the designed resolution.
6. Verify that the safety sequences are decoded correctly. Check if the payload data is decoded correctly. Check the resolution of the WAIT command. Verify that when the Command Code (CC = 11), the functional interface is dropped as this code is not used.
7. Verify that the right number of bytes are transmitted to the host by the Uart2 transmitter.

### 6.3.1. Inspection of signals in the Fault Emulation system

Consider the case where the communication bus is appropriately setup with the correct number of bytes and the fault campaign is run. Figure 6.14 depicts the onset of a stuck-at-1 fault at fault position 1. This waveform explains the fault sabotage signals. Take note of the three signals in this waveform: **EN**, **CS_n**, and **F1_CS_n**. The **EN** is the global enable signal that controls the activation and deactivation of the fault. **CS_n** is the original input node of a gate in the netlist. **F1_CS_n** is the modified input node of the gate. The power-on-reset ($por\_n$) is an active low reset to the Design Under Test. Once, this signal goes high, it indicates the start of execution of the functional interface. After the start of functional interface, the global fault enable signal is used to random activate the fault as shown in the waveform. Here stuck-at-1 fault is activated at 540 ns after the $por\_n$ signal goes high. As soon as the $EN$ goes high, the modified input node creates a fault in the netlist. It can be seen that after the fault enable is active the original input continues to change, whereas the modified input is stuck at 1. This verifies the third point in the verification that the fault is injected at the right location in the netlist.

Figure 6.15 displays the execution of a single transaction of UART containing 4 faults. Firstly, observe the signal **Uart_data_received** in the waveform, this goes high only when the right number of bits are received by the UART1 receiver on the FPGA. The next signal **Data_valid** is the output of the CRC block which validates that the received data is correct. Once the data is validated, the fault injection phase begins for fault site 1 which is indicated by the signal **fault_injection_start**. Once the fault is created, the **fault_injection_done** flag is set as shown in the figure. Now the functional and safety sequences are executed in the presence of fault at location 1, and the outputs are sent to the HDI master on the PC side over the UART2 transmitter. The execution of the functional interface is done when the variable **funct_intf_done** is set to high. This completes one cycle of fault injection and functional evaluation. Similarly, all the faults in the transaction are tested. Since, this example has only four faults in a single transaction, when this is complete, the **Frame_done** signal is set and this is notified to the HDI master that the transaction is complete and the HDI slave is ready for the next transaction. Similarly, all transactions are completed, and the execution time for the entire campaign is recorded. This is the performance figure targeted in this work.

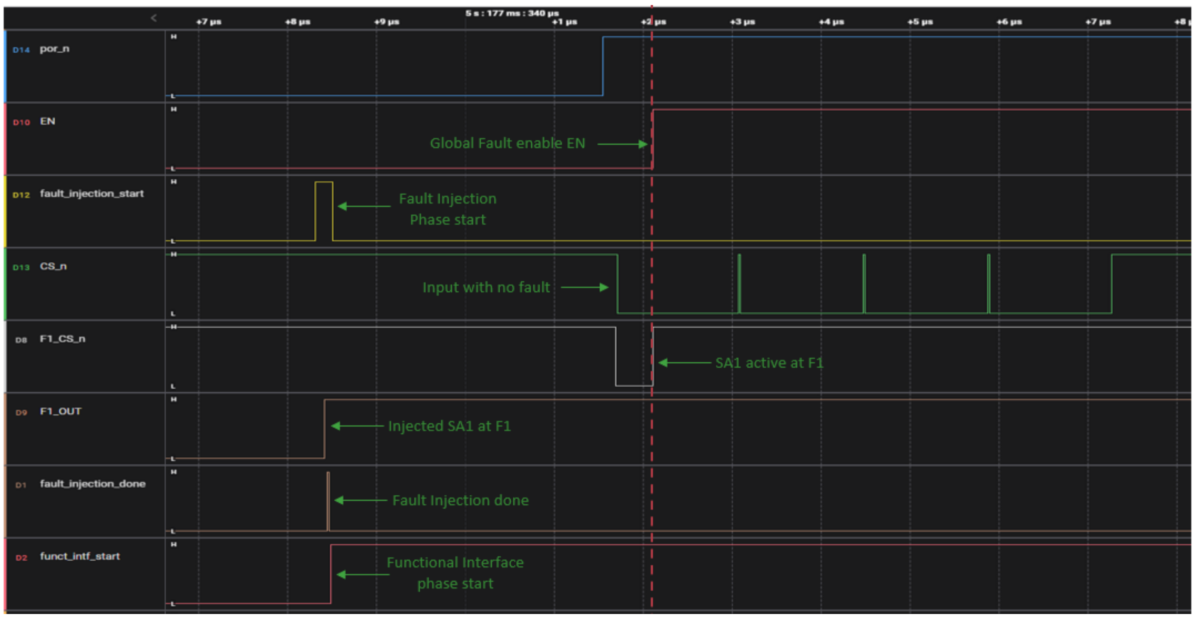


**Figure 6.14:** Create of a Stuck-at-1 fault at fault site 1

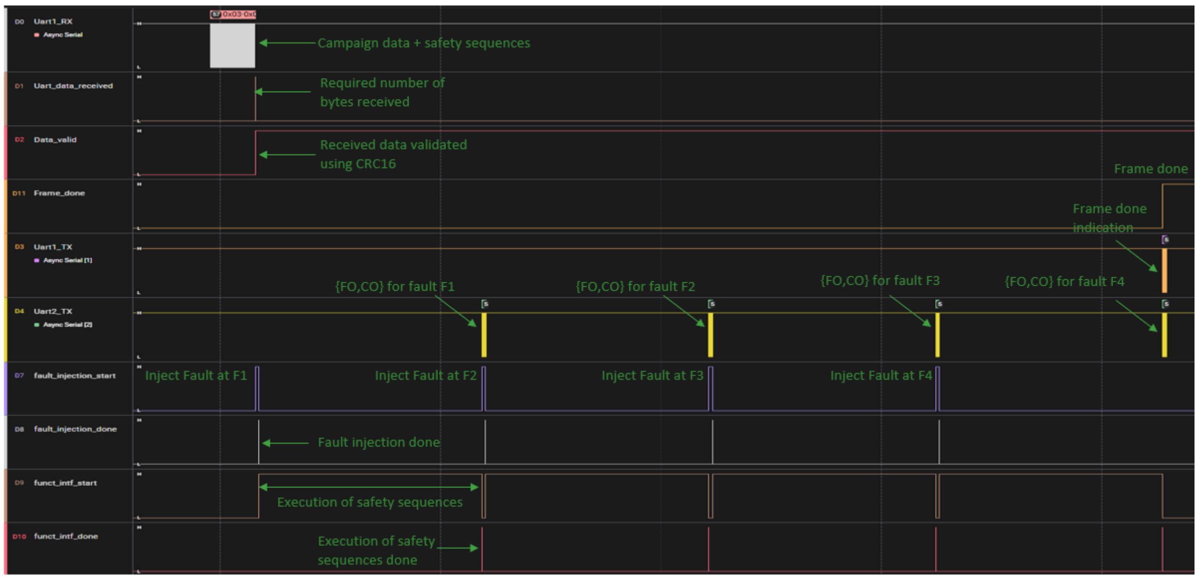


**Figure 6.15:** Example: execution of fault campaign for a single transaction of UART containing four faults

# 7 System Evaluation

This chapter introduces the Design Under Test for this work andsummarizes the performance of both emulation systems for the DUT and gives fault classification outcomes. It compares the two designs and their advantages as well as their disadvantages.

## 7.1. Design Under Test

For proof of concept, a simple system was implemented consisting of SPI serial communication protocol as shown in Figure 7.1 is designed. The logic is replicated here to create redundancy in order to identify all single-point failures by comparing one SPI interface against another. The Safety Mechanism (SM) in this example is the XOR gate, which detects the difference in the outputs of the functional and safety SPI interfaces and flags their mismatch as the checker output.

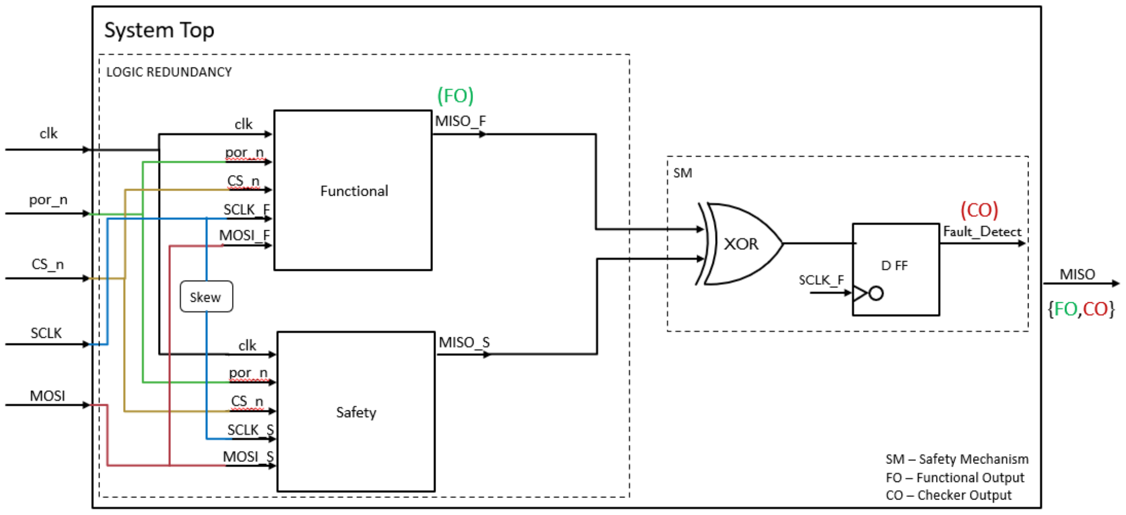


**Figure 7.1:** Overview of DUT implementing the logic redundancy safety strategy and XOR-based Safety Mechanism

The detailed block diagram of the Design Under Test consisting of the SPI protocol and register file to perform register write/read is shown in Figure 7.2. The register file has 5 address spaces and can be accessed for read/write using the $addr$ variable of width 3 bits. The first four spaces $addr = \{0, 1, 2, 3\}$ are used to perform functional read/write and the last address space with $addr = 4$ is reserved for the checker output.

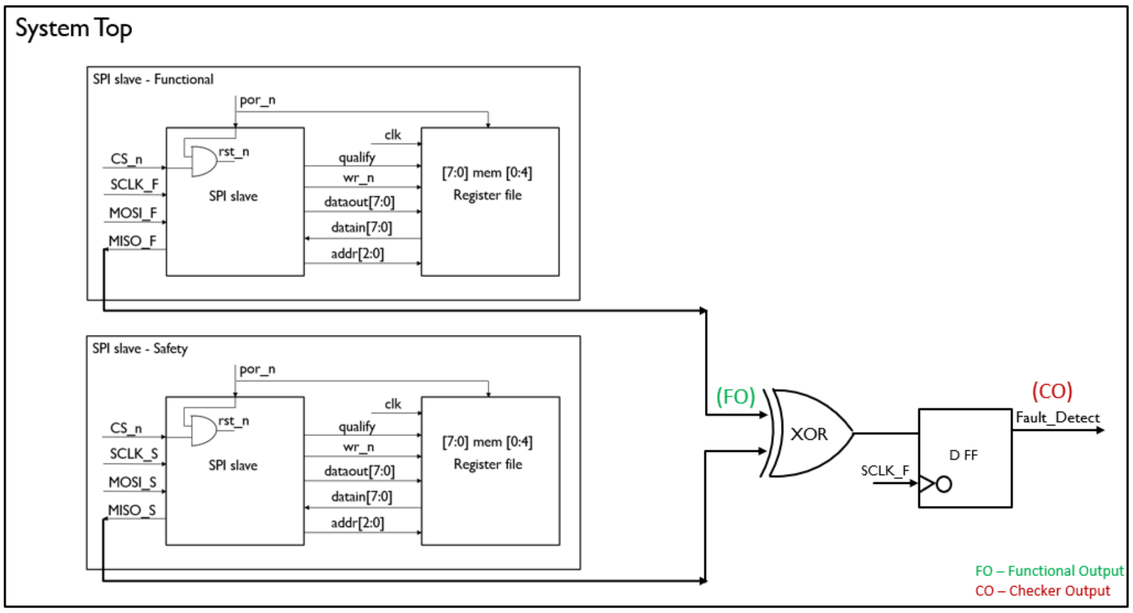


**Figure 7.2:** Overview of the Design Under Test

The SPI master is used to interface with the DUT; the functional interface between the SPI master and the DUT is depicted in Figure 7.3.

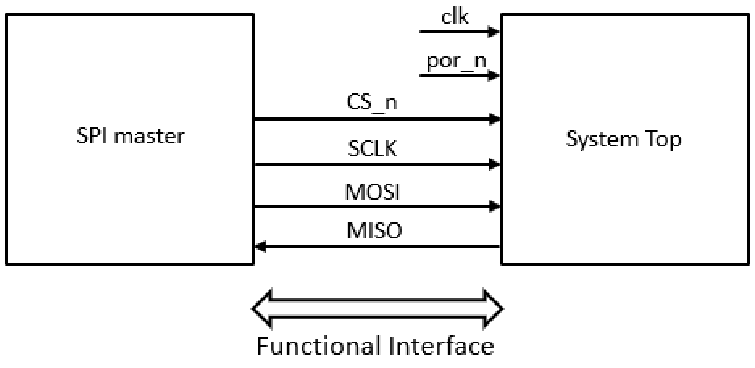


**Figure 7.3:** Functional Interface to the DUT

The Safety Sequence (SS) used to detect the single-point faults for this example is as follows,

1. Write 8-bit first checkerboard data 8'hAA to register with addr = {0,1,2,3}.
2. Read data from the address space previously written and compare it to 8'hAA. This is the first functional output FO1.
3. Write 8-bit second checkerboard data 8'h55 to register with addr = {0,1,2,3}.
4. Read data from the address space previously written and compare it to 8'h55. This acts as the second functional output FO2.
5. Read data from address space addr = 4. This is the checker output (CO). If there is a fault in the system and it is detected, this output will have logic one, the output will be zero.
6. Clear the register with checker output for the execution of the next functional sequence by writing 1 to the register with addr = 4.

Both the Functional Output (FO) and the Checker Output (CO) are read through the functional part SPI output MISO_F highlighted in all the waveforms. These two outputs are used for fault classification. First, the design is simulated without any fault, this is known as the golden run. The simulation output for the golden run is shown in the waveform in Figure 7.4.

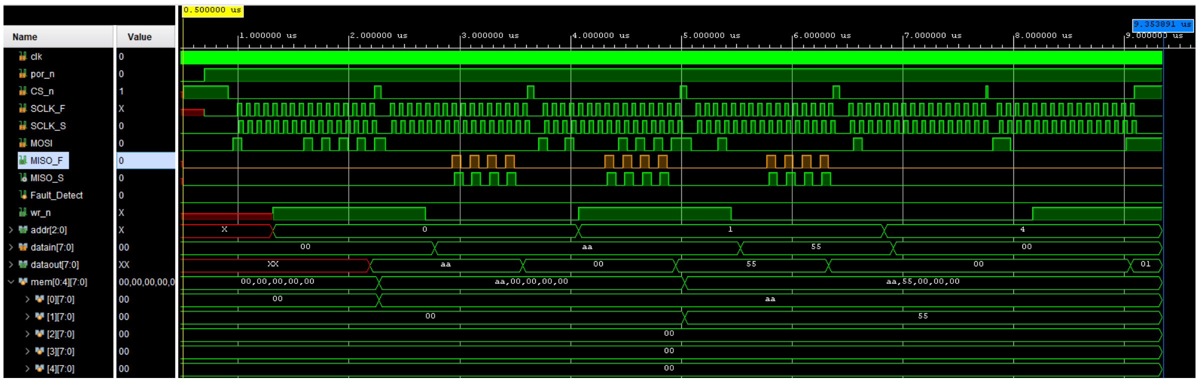


**Figure 7.4:** Simulation output for golden run, FO1 = 8'hAA, FO2 = 8'h55, CO = 8'h00

To evaluate the design for the detection of the four types of flaws, Detected Dangerous, Detected Non-Dangerous, Undetected Dangerous, and Undetected Non-Dangerous, four distinct scenarios are produced, as illustrated below,

1. The **Undetected Non-Dangerous (UU)** fault can be generated by injecting fault in the Safety Mechanism as shown the Figure 7.5. If Stuck-at-0 fault is introduced at $Fault\_Detect$ signal in the Design. This fault does not affect the functionality of the system and hence is unharmful.
2. The **Detected Dangerous (DD)** fault can be created by injecting SA1 fault in one of the data bits while performing register write. This is shown in Figure 7.6. From the waveform, it can be observed that the FO1 is not the same as in the golden run and this is detected as the CO = 3.In this case, the system can be taken to a safe state.
3. The **Non-Dangerous Detected (UD)** fault can be produced by injecting fault in the safety part of the system as shown in Figure 7.7. Here both FO1 and FO2 are the same as in the golden run, however, the fault in the safety part is detected by the SM. Since the fault is detected, it is unharmful as the functionality is unaffected.
4. The **Dangerous Undetected (DU)** fault is created by tying the $wr\_n$ signal of the functional part to zero as shown in Figure 7.8. This creates a fault in the read part of the checker output. This shows that the DUT is not designed with adequate safety levels and extra strategies must be implemented to enhance the safety of the design, where this fault does not go undetected. This is the most dangerous type of fault. The goal is to design the system with as less of these faults as they can lead to hazardous events. The primary requirement for certifying a product with ASIL-D is to show evidence that there are less than 1% of DU faults. However, the objective of this project is not to test the most safe design but to build a platform that can detect all type of faults.

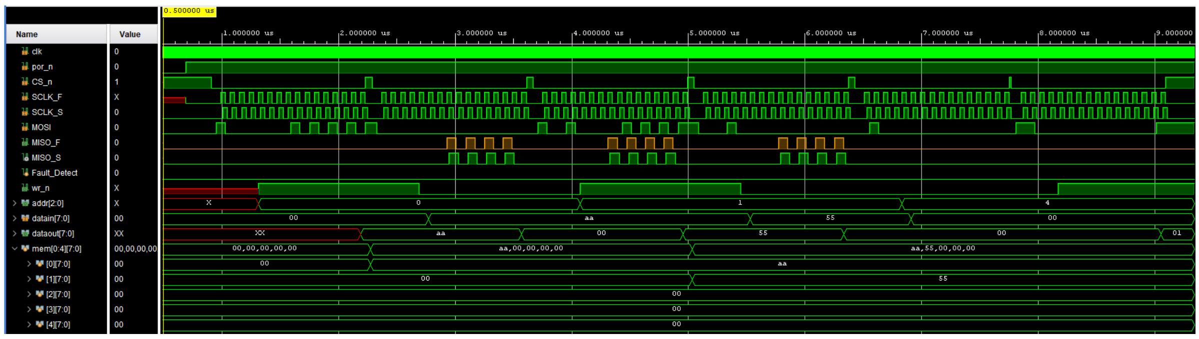


**Figure 7.5:** Simulation output for fault type Undetected Non-Dangerous(UU), FO1 = 8'hAA, FO2 = 8'h55, CO = 8'h00

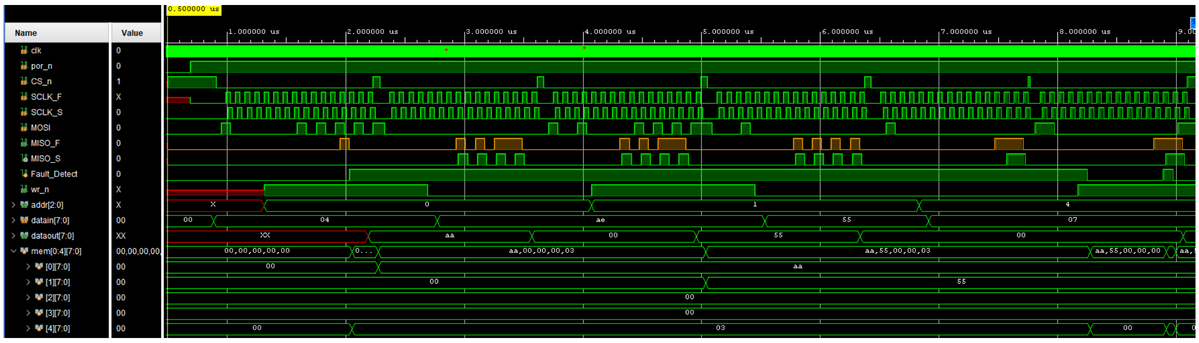


**Figure 7.6:** Simulation output for fault type Detected Dangerous(DD), FO1 = 8'hAE, FO2 = 8'h55, CO = 8'h03

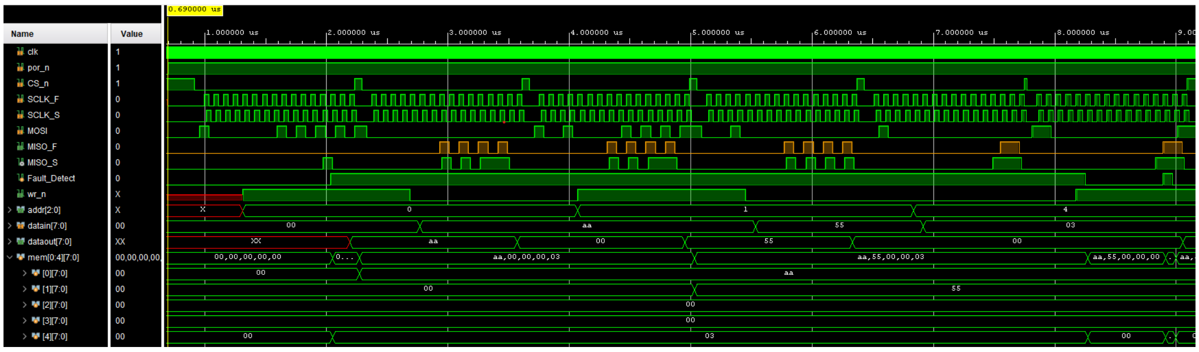


**Figure 7.7:** Simulation output for fault type Detected Non-Dangerous (UD), FO1 = 8'hAA, FO2 = 8'h55, CO = 8'h03

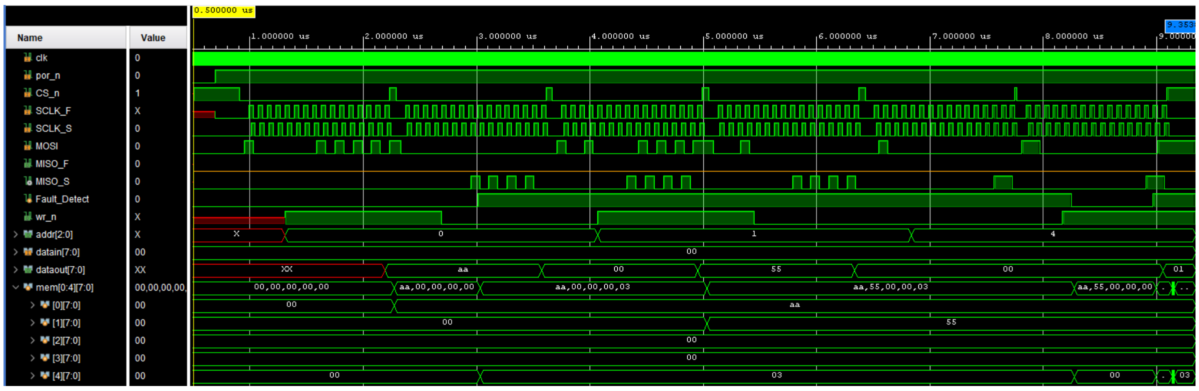


**Figure 7.8:** Simulation output for fault type Undetected Dangerous (DU), FO1 = 8'h00, FO2 = 8'h00, CO = 8'h00

The classification of faults is done by comparing the faulty run against the golden simulation as shown in table 7.1. In this example, there are two functional outputs FO1 and FO2, so the functional output is determined as

$$FO = FO1_{faulty} \neq FO1_{golden} \mid FO2_{faulty} \neq FO2_{golden} \tag{7.1}$$

and the checker output is determined as

$$CO \neq 0 \tag{7.2}$$

| | Golden run | UU {FO,CO} = {0,0} | DD {FO,CO} = {1,1} | UD {FO,CO} = {0,1} | DU {FO,CO} = {1,0} |
|---|---|---|---|---|---|
| FO1 | 8′hAA | 8′hAA | 8′hAE | 8′hAA | 8′h00 |
| FO2 | 8′h55 | 8′h55 | 8′h55 | 8′h55 | 8′h00 |
| CO | 8′h00 | 8′h00 | 8′h03 | 8′h03 | 8′h00 |
| FO (FO1 \| FO2) | 0 | 0 | 1 | 0 | 1 |
| CO | 0 | 0 | 1 | 1 | 0 |

**Table 7.1:** Classification of faults

## 7.2. Performance of Purely Programmable-based Architecture

The DUT designed for this experiment was tested using the first architecture that was introduced in Figure 6.1. The communication interface as explained in section 6.1 transfers fault information from the Host Debug Interface (HDI) master on the PC side to the Host Debug Interface (HDI) slave on the FPGA. However, while running the fault campaign, this transfer comes with a software overhead as shown in Figure 7.9. There is an overhead of $\sim 12ms$ between the successive transfers. This adds a huge latency to the execution time of the fault campaign.

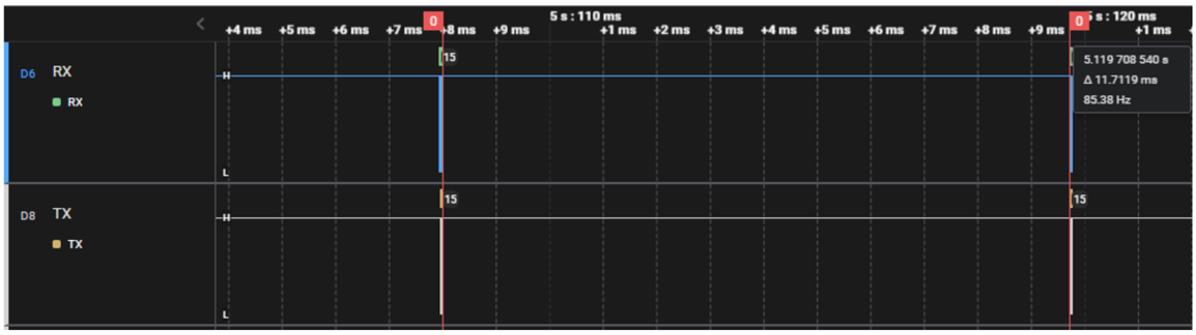


**Figure 7.9:** Software overhead during the transfer of fault information using the communication interface

Moreover, the execution of safety sequences using channel B as shown in Figure 6.1 also contains an overhead of $\sim 7ms$ between two successive SPI commands as shown in Figure 7.10.

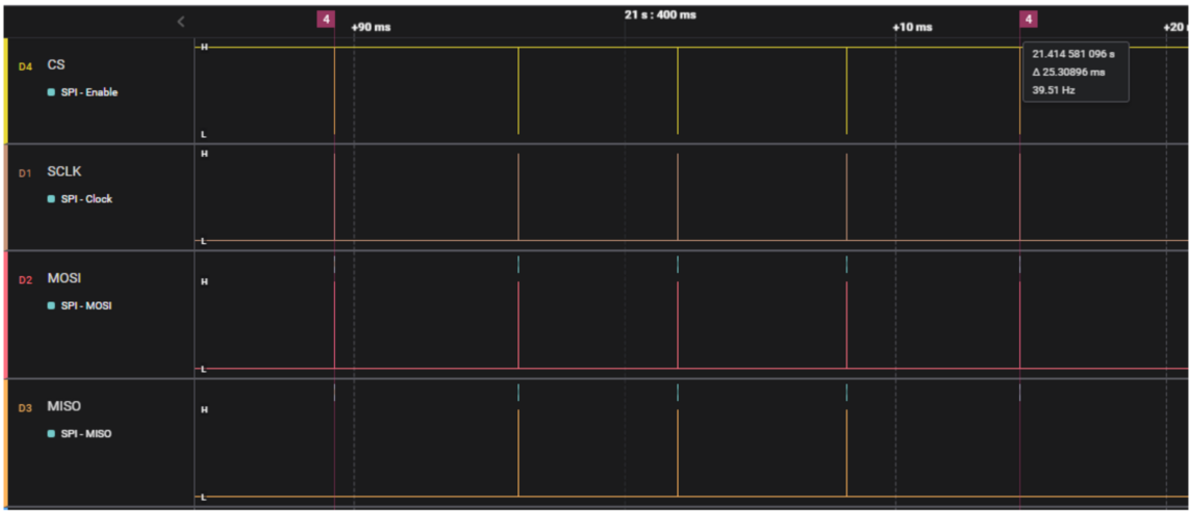


**Figure 7.10:** Software Overhead during execution of the safety sequences

Taking into account all the software overhead, the table 7.2 displays the various parameters involved in running the campaign, and the anticipated time is formulated.

| Symbol | Value | Unit | Description |
|---|---|---|---|
| $t_{rtd}$ | $42 \times 10^{-3}$ | s | Realtime duration of a functional and safety patterns |
| $N_{faults}$ | 640K | | Total number of faults in the campaign |
| $t_{single}$ | $20 \times 10^{-3}$ | s | Time taken to transfer information of a single fault from the host PC to the FPGA |
| $t_{fclock}$ | $5 \times 10^{-9}$ | s | Periof of clock for injecting the fault at the desired location |

**Table 7.2:** Campaign configuration parameters for Architecture 1

The total execution for performing the fault campaign of size 1M faults using architecture 1 can be formulated as below,

$$\begin{aligned} T_{total} &= N_{faults}(\times t_{ci} + t_{fclock} \times N_{faults} \times 2) + t_{rtd} \\ &= 640000 \times 20 \times 10^{-3} + (5 \times 10^{-9} \times 640000 \times 2) + 42 \times 10^{-3} \\ &= 4.38 \times 10^{4} seconds \\ &= 12.22 hours \end{aligned} \tag{7.3}$$

Out of the total execution time, the software accounts for roughly 10 hours which is 85% of the total time. Therefore, the execution time does not meet the requirements of this project. Thus, in Architecture 2, the limitations of this design were recognized and improved. The performance of the redesigned architecture will be presented in the next section.

## 7.3. Performance using Architecture 2

The major software overheads that occurred in Architecture 1 were overcome in this design as discussed in section 6.2. In this design, the overhead caused by the communication interface between the PC and the FPGA was distributed by delivering fault information for more than one fault at a time. This lowered the overhead incurred for each defect. The optimal number of faults per UART frame was determined to be 1023 faults as explained in the section 6.2. The time taken to perform fault injection for 1023 faults and run the safety sequence illustrated in section 7.1 is displayed in Figure 7.11. The different parameters involved in the fault campaign for this architecture are explained in table 7.3.

| Symbol | Value | Unit | Description |
|---|---|---|---|
| $N_{faults}$ | 640K | | Total number of faults in the campaign |
| $n_{single}$ | 1023 | | Faults per frame of Uart |
| $n_{frames}$ | $\frac{N_{faults}}{n_{single}}$ | | Number of Uart frames |
| $t_{rtd}$ | $10 \times 10^{-2}$ | s | Realtime duration of a functional and safety patterns |
| $t_{single}$ | $20 \times 10^{-3}$ | s | Time to transmit information of 1023 faults + safety sequence from the host PC to the FPGA using Uart |
| $t_{fclock}$ | $5 \times 10^{-9}$ | s | Period of the clock for injecting the fault at the desired location |

**Table 7.3:** Campaign configuration parameters for Architecture 2

The total execution time for performing the single-point and transient fault campaign with a fault space of 640K faults is calculated as shown below,

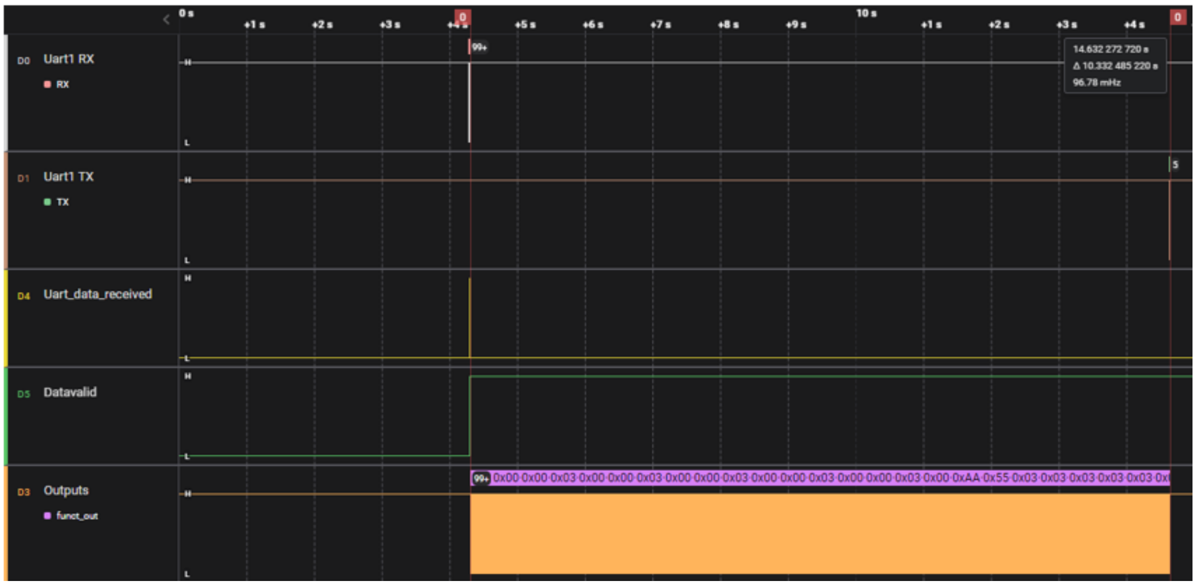


**Figure 7.11:** Time taken to execute the single frame of Uart consisting of 1023 faults

$$
\begin{aligned}
T_{total(SPF,TF)} &= n_{frames} \times n_{single}(\frac{t_{single}}{n_{single}} + (t_{flock} \times N_{faults} \times 2) + t_{rtd}) \\
&= 626 \times 1023(\frac{20 \times 10^{-3}}{1023} + (5 \times 10^{-9} \times 640K \times 2) + 10 \times 10^{-3}) \\
&= 6.482 \times 10^{3} seconds \\
&= 1.8 hours \\
T_{overhead640Kfaults} &= 23 ms \\
T_{overheadperfault} &= 120 us
\end{aligned}
\tag{7.4}
$$

## 7.4. Fault Classification Results

Fault classification diagnostics are required as evidence to certify integrated circuits (ICs) for Automotive Safety Integrity Level (ASIL) in the context of automotive safety-critical applications. ASIL is a risk classification system defined in the ISO 26262 standard as explained in section 2.1.1, which is used in the automotive industry to assess and manage the safety of electronic systems. Fault classification diagnostics are essential because they give precise information on the nature and behavior of faults that may arise in integrated circuits (ICs) used in safety-critical systems. This data is critical for calculating the possible impact of errors on system safety and building suitable security mechanisms. Furthermore, these safety metrics demonstrate to consumers that design methods like redundancy, error detection and correction, and other safety measures are effective in providing the greatest degree of safety and dependability in automotive electronics.

The fault classification is performed using the two outputs, namely the functional and checker output as shown in Figure 7.12. The main goal is to capture most of the Dangerous Undetected faults. For instance, for ASIL-D design, the Dangerous Undetected faults must be less than 1% of the total fault population. If the diagnostic coverage of the fault classification is not according to the expected ASIL level required, this activity helps in improving the safety mechanisms implemented in the design.

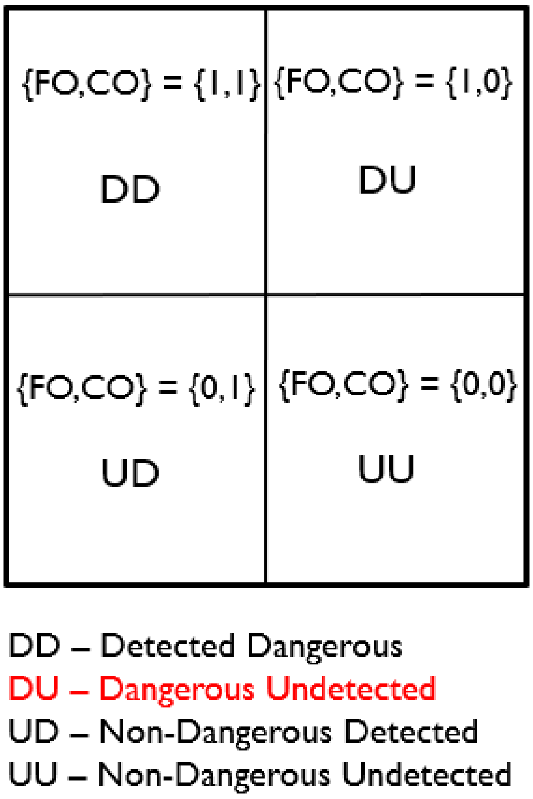


**Figure 7.12:** Fault classification matrix

The final output of this experiment is the fault classification for the Design Under Test and is displayed in Table 7.4. Some random nodes from the netlist were selected showcasing all the four type of faults.

| Fault_site | Input Node | Modified Node | Fault Type |
|---|---|---|---|
| 1 | CS_n | F1_CS_n | DD |
| 2 | n_629 | F3_n_629 | DD |
| 3 | n_628 | F5_n_628 | UD |
| 4 | n_626 | F14_n_626 | UU |
| 5 | n_627 | F25_n_627 | DU* |
| 6 | n_624 | F76_n_624 | UU |

**Table 7.4:** Fault Classification

The Fault site indicates the location of the fault in the long fault chain, the Input node is the input of a gate in the netlist, the modified node is the node where the fault is created and the last category is the fault type. It can be seen that the system is capable of detecting all four types of faults. As previously stated, the goal of this experiment was not to test the safest design, but to provide an FPGA-based fault emulation platform where the design could be evaluated for all forms of faults. Therefore, this goal is met.

## 7.5. Performance Results

The main objective of this project is to provide a fault emulation platform that aids in the acceleration of fault injection activities as compared to the simulation-based technique. For a fair comparison to the simulation benchmark introduced in section 3.3, the fault campaign was run for the same fault space of 640K faults. The real-time duration of the execution of the functional interface was also set as per the benchmark to $10 \times 10^{-2}s$. The performance of architectures 1 and 2 in comparison to the benchmark is shown in Table 7.5.

| Description | Unit | Value | Value | Architecture 1 | Architecture 2 |
|---|---|---|---|---|---|
| Max cores available for Digital Fault Injection | | 160 | 50 | | |
| Realtime duration of a functional pattern, including classification | s | $10^{-2}$ | $10^{-2}$ | $3.5 \times 10^{-2}$ | $10^{-2}$ |
| Number of faults | | 640K | 640K | 640K | 640K |
| Single DMS simulation duration | s | $3 \times 10^2$ | $3 \times 10^2$ | | |
| Campaign duration (w/o overhead) (overhead ~ 50%) | s | $6 \times 10^5$ | $1.92 \times 10^6$ | $4.38 \times 10^4$ | $6.482 \times 10^3$ |
| Campaign duration | days | 6.94 | 22.22 | 12.22 hours | 1.8 hours |

**Table 7.5:** Performance comparison of Architecture 1 and 2 against the simulation benchmark

## 7.6. Limitations of the Fault Emulation System and Future Work

This project comprises several components, including design preparation and mapping to the FPGA, hardware development for the fault manager, and software implementation for communication between the host PC and the FPGA. Thus, to achieve the completion of the task within the limited time frame, some simplifications were granted by the stakeholders. These are listed below with their implications and how these can be achieved in future work.

1. During gate-level synthesis, five simplifications were made to expedite the development of this platform: Netlist Flattening, No DFT option, No clock-gated design for evaluation, No faults tested in the clock tree, and limited cells in the technology library. These options did not affect the evaluation of the performance of fault campaign. However, it greatly eased the process of mapping the ASIC gate-level netlist onto the FPGA with fault injectors. To avoid these simplifications, further work will necessitate system improvements. In reality, the actual netlist is hierarchical; this will necessitate an enhancement of the Python-based compiler to recognize and interpret hierarchical netlists. Second, to integrate DFT, an additional step will be necessary during the netlist translation from ASIC to FPGA flow. Because FPGAs do not contain scan flops by default, a special design of the scan flop is necessary, as well as a method of inserting them into the netlist. To evaluate the Design Under Test consisting of gated clocks, special design strategies need to be implemented to bring the design into the FPGA. Lastly, the Python script used for the translation of netlist must be improved to include all the standard cells in the technology library.

2. Whenever a new system is designed, it requires extensive verification to eliminate the possibility of systematic errors. Typically a V-model-based verification process is followed in verifying the system. However, due to the limited time frame, the platform developed was verified using waveforms and simple test cases for different scenarios. This activity was limited to critical points and was not done exhaustively. In the future, this activity can be expanded to create a structural verification environment using SystemVerilog and UVM.

3. The fault emulation platform requires post-processing of the outputs. In this work, a simple Python-based setup was implemented to perform the fault classification and record the diagnostic metrics. However, this work can be expanded to include a more sophisticated GUI-based platform to report the ASIL determination and safety metrics.

4. Today most of the systems are mixed-signal in nature consisting of both Analog and Digital parts. In this experiment, only a purely digital circuit was tested, however, the emulation platform is designed in a way to support the Analog Interface. Thus, in future work, complex systems consisting of analog models using EEnet, Real Number Modeling, and Wreal datatype along with the digital parts can be designed and tested on the emulation system.

# 8 Conclusion

Today, electric cars are dominating the automotive market. When designing ICs deployed in these cars, safety is one of the primary concerns. In this project, the various steps involved in designing the chip for safety-related purposes were explored. Fault injection is one of the major activities performed during the production of such ICs to provide evidence in order to certify the product with ASIL level. The safety metrics of this activity also serve as a proof to the OEMs that the safety measures implemented in the design are sufficient.

Currently, a simulation-based method is used for performing fault injection. This has certain limitations, mainly, a large execution time for running the fault campaigns, availability of tool licenses and its cost, and the need for reduction of fault space as designs grow complex. For this experiment, the performance numbers for Battery Management System (BMS) IC developed at Analog Devices were taken into account for comparison. For a fault space of 640K faults, the time required to accomplish the fault injection activity was recorded to be 22 days.

Thus, to overcome the above challenges, various fault injection techniques were explored during the literature study of this research and it was concluded that FPGA-based fault emulation is one of the most feasible and cost-effective ways to carry out this activity. Based on the limitations and requirements, detailed system requirements and specifications for the emulation-based platform were formulated. The three major characteristics prioritized are fault campaign execution speed, hardware cost-effectiveness, and system scalability. Following that, the project was divided into three major sections. The Pre-processing process prepared the Design Under Test for FPGA testing. This comprised gate-level synthesis to transform the RTL design to a gate-level netlist and then modifications were made to include custom-designed fault saboteurs at the gate nodes in the netlist. Then the logic equivalence check was between the DUT and the modified netlist to ensure that the functionality of the design was not lost in the alterations. Now, the design is ready for testing on the FPGA.

The second main task was to develop the hardware surrounding the DUT to control the fault campaign. The fault manager consisted of two main elements. First, the Host Debug Interface (HDI) master was implemented on the host PC side whose job was to collect all the campaign-related parameters from the user, prepare the communication bus, and transmit the data to the Host Debug Interface (HDI) slave on the FPGA. HDI slave is responsible for supervising the entire fault campaign. It consists of two sub-blocks, namely, the Fault Injection System and the Functional Decoder System. After receiving the campaign data from the HDI master, it instructed the Fault injector to perform the injection of fault one at a time. The safety sequence decoder then sends the data to the SPI master to execute the functional and safety sequences. The outputs of the DUT were recorded back by the decoder and then put onto the FIFO to be transmitted out of the FPGA back to the HDI master. The final task of the HDI master was to use the functional and checker output and provide the fault classification which is the expected result of this experiment. These diagnostics serve as evidence for the ASIL certification of the product.

In this work, two types of architectures were experimented and the total execution time for running the fault campaign including the software overheads were compared. The first architecture provided a performance of roughly 13 hours to run the fault campaign of the size 640k faults. The large execution time was due to the software overhead in the communication and functional interface. These shortcomings were improved in architecture 2 which gave a performance of 1.8 hours for the same fault population size. Similarly, with a fault space of 1M faults, the fault campaign execution duration was 2.8 hours. Therefore, the target execution time of less than 3 hours is met. As a result, a 296x speedup was achieved as compared to the simulation-based fault injection approach.

# References

[1] ISO 3779. “Road vehicles — Vehicle identification number (VIN) — Content and structure”. In: 2018.

[2] Seyed-Nematollah Ahmadian and Seyed-Ghassem Miremadi. “Fault injection in mixed-signal environment using behavioral fault modeling in Verilog-A”. In: *2010 IEEE International Behavioral Modeling and Simulation Workshop*. 2010, pp. 69–74. DOI: 10.1109/BMAS.2010.6156601.

[3] David de Andres et al. “Fault Emulation for Dependability Evaluation of VLSI Systems”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.4 (2008), pp. 422–431. DOI: 10.1109/TVLSI.2008.917428.

[4] L. Antoni, R. Leveugle, and B. Feher. “Using run-time reconfiguration for fault injection applications”. In: *IMTC 2001. Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference. Rediscovering Measurement in the Age of Informatics (Cat. No.01CH 37188)*. Vol. 3. 2001, 1773–1777 vol.3. DOI: 10.1109/IMTC.2001.929505.

[5] Felipe Augusto da Silva et al. “Combining Fault Analysis Technologies for ISO26262 Functional Safety Verification”. In: *2019 IEEE 28th Asian Test Symposium (ATS)*. 2019, pp. 129–1295. DOI: 10.1109/ATS47505.2019.00024.

[6] Ghada Bahig and Amr El-Kadi. “Formal Verification of Automotive Design in Compliance With ISO 26262 Design Verification Guidelines”. In: *IEEE Access* 5 (2017), pp. 4505–4516. DOI: 10.1109/ACCESS.2017.2683508.

[7] Emmanuel Blot. *PyFtdi*. https://eblot.github.io/pyftdi/. 2001-2020.

[8] M. Brandl et al. “Batteries and battery management systems for electric vehicles”. In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, pp. 971–976. DOI: 10.1109/DATE.2012.6176637.

[9] L. Burgun et al. “Serial fault emulation”. In: (1996), pp. 801–806. DOI: 10.1109/DAC.1996.545681.

[10] C. C. Chan. “The State of the Art of Electric, Hybrid, and Fuel Cell Vehicles”. In: *Proceedings of the IEEE* 95.4 (2007), pp. 704–718. DOI: 10.1109/JPROC.2007.892489.

[11] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. “Fault emulation: a new approach to fault grading”. In: *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. 1995, pp. 681–686. DOI: 10.1109/ICCAD.1995.480203.

[12] P. Civera et al. “Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits”. In: (2001), pp. 250–258. DOI: 10.1109/DFTVS.2001.966777.

[13] P. Civera et al. “FPGA-based fault injection for microprocessor systems”. In: *Proceedings 10th Asian Test Symposium*. 2001, pp. 304–309. DOI: 10.1109/ATS.2001.990301.

[14] *Conformal Smart LEC*. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/conformal-smartlec-ds.pdf.

[15] *Cyclic redundancy check*. https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[16] Stefano Di Carlo et al. “A fault injection methodology and infrastructure for fast single event upsets emulation on Xilinx SRAM-based FPGAs”. In: *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2014, pp. 159–164. DOI: 10.1109/DFT.2014.6962073.

[17] A. P. Dorey et al. “Introduction to VLSI Testing”. In: *Rapid Reliability Assessment of VLSICs*. Boston, MA: Springer US, 1990, pp. 1–14. ISBN: 978-1-4613-0587-3. DOI: 10.1007/978-1-4613-0587-3_1. URL: https://doi.org/10.1007/978-1-4613-0587-3_1.

[18] Denis Dutey et al. “Prevention and Detection Methods of Systematic Failures in the Implementation of SoC Safety Mechanisms not Covered by Regular Functional Tests”. In: *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2021, pp. 87–92. DOI: 10.1109/DDECS52668.2021.9417073.

[19] EDN. *Redundancy for safety-compliant automotive other devices*. https://www.edn.com/redundancy-for-safety-compliant-automotive-other-devices/. 2014.

[20] *Electric car sales break new records with momentum expected to continue through 2023.* https://www.iea.org/energy-system/transport/electric-vehicles. 2023.

[21] Mohammad Eslami et al. "A survey on fault injection methods of digital integrated circuits". In: *Integration* 71 (2020), pp. 154–163. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.11.006. URL: https://www.sciencedirect.com/science/article/pii/S016792601930402X.

[22] Umer Farooq. "Pre-Silicon Verification Using Multi-FPGA Platforms: A Review". In: *Journal of Electronic Testing* 37 (Feb. 2021), pp. 1–18. DOI: 10.1007/s10836-021-05929-1.

[23] Kim R. Fowler. "Chapter 1 - Introduction to Good Development". In: *Developing and Managing Embedded Systems and Products*. Ed. by Kim R. Fowler and Craig L. Silver. Oxford: Newnes, 2015, pp. 1–38. ISBN: 978-0-12-405879-8. DOI: https://doi.org/10.1016/B978-0-12-405879-8.00001-5. URL: https://www.sciencedirect.com/science/article/pii/B9780124058798000015.

[24] "FPGA-based switch-level fault emulation using module-based dynamic partial reconfiguration". In: *Microelectronics Reliability* 48.10 (2008), pp. 1724–1733. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2008.06.003. URL: https://www.sciencedirect.com/science/article/pii/S0026271408001297.

[25] *Genus Synthesis Solution*. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/genus-synthesis-solution-ds.pdf.

[26] Nikolaos Georgoulopoulos, Athanasios Mekras, and Alkiviadis Hatzopoulos. "Design of a SystemVerilog-Based VCO Real Number Model". In: *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 2019, pp. 1–4. DOI: 10.1109/MOCAST.2019.8741782.

[27] Gergely Hantos, David Flynn, and Marc P. Y. Desmulliez. "Built-In Self-Test (BIST) Methods for MEMS: A Review". In: *Micromachines* 12.1 (2021). ISSN: 2072-666X. URL: https://www.mdpi.com/2072-666X/12/1/40.

[28] Jin-Hua Hong, Shih-Arn Hwang, and Cheng-Wen Wu. "An FPGA-based hardware emulator for fast fault emulation". In: *Proceedings of the 39th Midwest Symposium on Circuits and Systems*. Vol. 1. 1996, 345–348 vol.1. DOI: 10.1109/MWSCAS.1996.594168.

[29] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157.

[30] Zih-Ming Huang et al. "FPGA-Based Emulation for Accelerating Transient Fault Reduction Analysis". In: *2022 IEEE 31st Asian Test Symposium (ATS)*. 2022, pp. 144–149. DOI: 10.1109/ATS56056.2022.00037.

[31] MOBILITY INSIDER. *What Is ASIL-D?* https://www.aptiv.com/en/insights/article/what-is-asil-d. 2020.

[32] Arpita Josef Zinner. "Analog Devices internal training on Automotive Functional Safety". In: 2022.

[33] Endri Kaja et al. "Fast and Accurate Model-Driven FPGA-based System-Level Fault Emulation". In: *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. 2022, pp. 1–6. DOI: 10.1109/VLSI-SoC54400.2022.9939615.

[34] P. Kenterlis et al. "A low-cost SEU fault emulation platform for SRAM-based FPGAs". In: *12th IEEE International On-Line Testing Symposium (IOLTS'06)*. 2006, 7 pp.-. DOI: 10.1109/IOLTS.2006.5.

[35] T. Kogan et al. "Advanced functional safety mechanisms for embedded memories and IPs in automotive SoCs". In: *2017 IEEE International Test Conference (ITC)*. 2017, pp. 1–6. DOI: 10.1109/TEST.2017.8242046.

[36] B.J Lavanyashree and S Jamuna. "Design of fault injection technique for VLSI digital circuits". In: (2017), pp. 1601–1605. DOI: 10.1109/RTEICT.2017.8256869.

[37] R. Leveugle et al. "Statistical fault injection: Quantified error and confidence". In: *2009 Design, Automation Test in Europe Conference Exhibition*. 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716.

[38] Chris Liechti. *pySerial*. https://pyserial.readthedocs.io/en/latest/pyserial.html. 2001-2020.

[39] C. Lopez-Ongil et al. "A Unified Environment for Fault Injection at Any Design Level Based on Emulation". In: *IEEE Transactions on Nuclear Science* 54.4 (2007), pp. 946–950. DOI: 10.1109/TNS.2007.904078.

[40] Celia Lopez-Ongil et al. "Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation". In: *IEEE Transactions on Nuclear Science* 54.1 (2007), pp. 252–261. DOI: 10.1109/TNS.2006.889115.

[41] Celia López-Ongil et al. "Techniques for Fast Transient Fault Grading Based on Autonomous Emulation". In: *CoRR* abs/0710.4757 (2007). arXiv: 0710.4757. URL: http://arxiv.org/abs/0710.4757.

[42] Akira Motohara and Hideo Fujiwara. "Design for Testability for Complete Test Coverage". In: *IEEE Design Test of Computers* 1.4 (1984), pp. 25–32. DOI: 10.1109/MDT.1984.5005686.

[43] Tiziano Munaro and Irina Muntean. "Early Assessment of System-Level Safety Mechanisms through Co-Simulation-based Fault Injection". In: *2022 IEEE Intelligent Vehicles Symposium (IV)*. 2022, pp. 1703–1708. DOI: 10.1109/IV51971.2022.9827327.

[44] Optima-DA. "An ISO 26262 Automotive Semiconductor Safety Primer". In: 2019.

[45] Terence Parr. *Conformal Smart LEC*. https://www.antlr.org/.

[46] Sarvesh Patankar et al. "Hybrid Methodology for Verification of SW Safety Mechanisms". In: *2021 IEEE 39th VLSI Test Symposium (VTS)*. 2021, pp. 1–4. DOI: 10.1109/VTS50974.2021.9441001.

[47] Ananthraj C R and Arnab Ghosh. "Battery Management System in Electric Vehicle". In: *2021 4th Biennial International Conference on Nascent Technologies in Engineering (ICNTE)*. 2021, pp. 1–6. DOI: 10.1109/ICNTE51185.2021.9487762.

[48] J. Raik et al. "Improved fault emulation for synchronous sequential circuits". In: *8th Euromicro Conference on Digital System Design (DSD'05)*. 2005, pp. 72–78. DOI: 10.1109/DSD.2005.50.

[49] Sebastian Reiter et al. "Fault injection ecosystem for assisted safety validation of automotive systems". In: *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. 2016, pp. 62–69. DOI: 10.1109/HLDVT.2016.7748256.

[50] "Road Vehicles—Functional safety— Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses". In: 2018, pp. 1–38.

[51] "Road vehicles—Functional Safety—Part 1: Vocabulary". In: 2018, pp. 1–47.

[52] Chantal Robach and Mathieu Scholive. "Simulation-Based Fault Injection and Testing Unsing the Mutation Technique". In: (2003). Ed. by Alfredo Benso and Paolo Prinetto, pp. 195–215.

[53] Alireza Rohani and Hans G. Kerkhoff. "Rapid transient fault insertion in large digital systems". In: *Microprocessors and Microsystems* 37.2 (2013). Digital System Safety and Security, pp. 147–154. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2012.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0141933112001652.

[54] Moga Natha Shankar Kumar and Karthikeyan Balakrishnan. "Functional Safety Development of Battery Management System for Electric Vehicles". In: *2019 IEEE Transportation Electrification Conference (ITEC-India)*. 2019, pp. 1–6. DOI: 10.1109/ITEC-India48457.2019.ITECINDIA2019-267.

[55] Felipe Augusto da Silva et al. "Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs". In: *2020 IEEE European Test Symposium (ETS)*. 2020, pp. 1–6. DOI: 10.1109/ETS48528.2020.9131568.

[56] Luca Sterpone and Massimo Violante. "A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs". In: *IEEE Transactions on Nuclear Science* 54 (2007), pp. 965–970.

[57] *The Many Facets of ISO 26262: Fault Injection Testing of Safety Critical Automotive Software*. https://www.embitel.com/blog/embedded-blog/fault-injection-testing-of-safety-critical-automotive-software. 2022.

[58] Gabriele Tomassetti. *Listeners And Visitors*. https://tomassetti.me/listeners-and-visitors/.

[59] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection". In: *2018 14th European Dependable Computing Conference (EDCC)*. 2018, pp. 1–8. DOI: 10.1109/EDCC.2018.00013.

[60] A. Ullah et al. "An FPGA-based dynamically reconfigurable platform for emulation of permanent faults in ASICs". In: *Microelectronics Reliability* 75 (2017), pp. 110–120. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2017.06.032. URL: https://www.sciencedirect.com/science/article/pii/S0026271417302202.

[61] Garazi Juez Uriagereka et al. "Fault injection method for safety and controllability evaluation of automated driving". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 1867–1872. DOI: 10.1109/IVS.2017.7995977.

[62] Audhild Vaaje. "Theorems for Fault Collapsing in Combinational Circuits". In: *J. Electron. Test.* 22.1 (Feb. 2006), pp. 23–36. ISSN: 0923-8174. DOI: 10.1007/s10836-006-6222-1. URL: https://doi.org/10.1007/s10836-006-6222-1.

[63] Wen X Wang L-T Wu C-W. *Vlsi Test Principles and Architectures: Design for Testability*. URL: http://site.ebrary.com/id/10169928.%20Accessed%20September%2013,%202023.

[64] *What is JTAG?* https://www.corelis.com/education/tutorials/jtag-tutorial/what-is-jtag.

[65] T. W. Williams. "Trends in Design for Testability". In: *Testing and Diagnosis of VLSI and ULSI*. Ed. by Fabrizio Lombardi and Mariagiovanna Sami. Dordrecht: Springer Netherlands, 1988, pp. 1–31.

[66] You Xiang et al. "Interactive Safety Analysis Framework of Autonomous Intelligent Vehicles". In: *MATEC Web of Conferences* 44 (Jan. 2016), p. 01029. DOI: 10.1051/matecconf/20164401029.

[67] Song Xu et al. "IC security evaluation against fault injection attack based on FPGA emulation". In: *2016 International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 285–288. DOI: 10.1109/FPT.2016.7929554.

[68] Hongchao Zheng, Long Fan, and Suge Yue. "FITVS: A FPGA-Based Emulation Tool For High-Efficiency Hardness Evaluation". In: *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. 2008, pp. 525–531. DOI: 10.1109/ISPA.2008.46.

[69] Haissam Ziade, Rafic Ayoubi, and R. Velazco. "A Survey on Fault Injection Techniques". In: *Int. Arab J. Inf. Technol.* 1 (Jan. 2004), pp. 171–186.