ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

School of Engineering

Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi" - DEI

Master Degree in Electronics for Intelligent Systems, Big data and Internet of things

MASTER THESIS

in

Automatic Insertion of Fault-injectable Flip-Flops in FPGA emulators of satellite computing platforms

Candidate
Athar Zafeer Kannamangalam
Aslam Basha

Supervisor
Prof. Francesco Conti

Co-Supervisor
Yvan Tortella

Academic year 2022/23
Session 3rd

எவ்வ துறைவது உலகம் உலகத்தோடு
அவ்வ துறைவ தறிவு.
-திருவள்ளுவர்

To live as the world lives, is wisdom
-Thiruvalluvar

Sommario

In un'era in cui il ritmo dell'esplorazione spaziale sta accelerando, la domanda di processori elettronici in grado di dimostrare affidabilità nel difficile ambiente spaziale è aumentata in modo significativo. Una delle principali preoccupazioni per i processori che operano nello spazio è la loro suscettibilità ai guasti indotti dalla radiazione di fondo dello spazio. Le particelle ad alta energia che colpiscono qualsiasi parte del processore possono causare errori nei dati. Gli approcci tradizionali, come l'indurimento delle radiazioni, incontrano sfide sostanziali, tra cui costi elevati e limitazioni nella fabbricazione. L'uso di processori COTS (Commercially Off-The-Shelf) rappresenta una potenziale soluzione. I recenti progressi tecnologici hanno reso i processori COTS non solo estremamente veloci ed economici, ma anche facilmente disponibili. Questo lavoro introduce un nuovo approccio attraverso lo sviluppo e l'integrazione di un modulo iniettore di guasti nell'architettura PULP (Parallel Ultra-Low Power). Ciò consente la simulazione e la valutazione dei guasti indotti dalle radiazioni, in particolare degli errori singoli e dei guasti bloccati. Valutando la tolleranza agli errori di questi processori, questa ricerca mira a aprire la strada a una metodologia per migliorarne l'affidabilità per le applicazioni spaziali. Sebbene i processori COTS non siano intrinsecamente progettati per le impegnative condizioni dello spazio, questo studio dimostra che con modifiche mirate e test rigorosi, possono essere adattati efficacemente per l'esplorazione spaziale. Questo approccio implica uno spostamento verso l'utilizzo di soluzioni più convenienti e tecnologicamente avanzate, rendendo i processori COTS utilizzabili per applicazioni spaziali critiche.

Abstract

In space exploration, the demand for electronic processors capable of demonstrating reliability in the harsh space environment has significantly increased. One of the main concerns for processors operating in space is their susceptibility to faults induced by radiation. High-energy particles striking any part of the processor may induce data errors. Traditional approaches, such as radiation hardening, encounter substantial challenges, including high costs and limitations in fabrication. The use of Commercially Off-The-Shelf (COTS) processors presents a potential solution. Recent advances have made Commercial Off-The-Shelf (COTS) processors widely accessible, offering speed and cost-efficiency. However, these generic COTS processors lack reliability features and are not tested for space applications. In addressing this, our approach integrates a fault injector module within a scalable processing platform, offering a tool that can be applied in early design phases. This facilitates validation against various faults, identifying vulnerable design areas, and enhancing overall system robustness without being limited to a specific architecture. This enables the simulation and evaluation of radiation-induced faults, notably single-event errors and stuck-at faults. While we utilize the PULP architecture as a test platform for our module, the applicability extends to a broader range of architectures. By assessing the fault tolerance of these processors, this research aims to pioneer a methodology for enhancing their reliability for space applications. Moreover, our tool provides insights into the parts of the design that are more sensitive than others, allowing us to identify areas requiring improvement to ensure reliability in space missions. This study demonstrates the potential for adapting COTS processors to space exploration, marking a shift towards more cost-effective and technologically advanced solutions viable for critical space-borne applications.

Contents

List of Figures

Chapter 1

Introduction

Venturing into the vast expanse of space, humanity has continuously sought to overcome the difficult challenges posed by its hostile environment. In the vacuum of space, beyond the Earth's atmosphere, electronic systems are bombarded with high-energy particles, leading to the phenomenon of single-event upsets and persistent stuck-at faults in the circuitry of space-bound processors. These anomalies have the potential to derail missions, destroy satellites, and endanger lives. Consequently, the field of space exploration has relied on specially crafted space-grade processors, engineered to withstand this bombardment through radiation-hardening techniques. While radiation-hardening techniques are effective, they come at a high cost both in terms of financial investment and technological advancement, often resulting in trade-offs between resilience and computational performance [3].

Enter the age of Commercial Off-The-Shelf (COTS) processors—devices that have revolutionized terrestrial applications with their performance and cost efficiency. Their potential utility in space applications remains largely untapped, primarily due to their inherent vulnerability to the relentless barrage of high-energy particles beyond Earth's protective atmosphere [3]. Yet, the allure of harnessing these processors for space exploration is undeniable. However, the architectures that grant them their impressive capabilities on Earth are also the cause of their failure in the face of cosmic radiation encountered in space. The integration of these commercially produced processors into space systems, therefore, necessitates a paradigm shift approach to equip them with

the necessary changes to operate reliably in the harsh extraterrestrial environment.

The work presented is a step in that direction. Through the design and integration of a fault injector into any processor architecture, this research demonstrates a method for simulating the effects of space-induced faults within terrestrial processors. This module injects errors into specific target locations, enabling the study of fault propagation pathways and the subsequent effect on the processor's outputs. The overarching aim is to utilize these insights to devise strategic modifications that enhance the reliability of COTS architecture, thereby extending their operationability to the domain of space exploration. This approach promises to bridge the gap between the resilience of space-grade processors and their desirability for high-performance, economical COTS systems. The modifications to these processors can be tailored to the specific demands of the application and the difficulties of the space environment in which they operate.

The implications of this research are far-reaching. By providing a detailed assessment of how COTS processors respond to injected faults, that mimic space conditions, it lays the groundwork for a new class of space-ready electronics. These enhanced processors could drastically reduce the cost and complexity of space missions, democratizing access to space technologies, and fostering innovation in the sector.

Chapter 2

Literature review

The need for viable and cost-effective electronic processors for space has been a cause of much research. Especially finding proper methodologies for testing COTS processors for their resilience to radiation-induced errors.

The survey by Ginosar [3] on processors for space applications presents a detailed overview of the resilience strategies employed to ensure the reliability of processors in the harsh conditions of space. This study helps to understand the critical balance between cost, performance, and radiation tolerance. It offers a good foundation for discussing various approaches that exist or are under research.

The study [1], underscores a holistic approach toward fault tolerance by weaving together hardware and software strategies across various system layers. A particular interest of this research is their testing strategy, which evaluates the impact of soft errors through extensive fault injection experiments. The tools used in this work for fault injection are the BEE3 FPGA and mixed simulation using supercomputers. This work required hefty resources to gather data on the fault resilience of the processor. Our work makes use of the specially-made fault injection module which can be incorporated into any processor architecture to do simulations of radiation-induced faults.

One of the methodologies to inject errors into a processor design for testing its resilience is using software. In the work [5], the validation of the design is primarily

done using the software-based error injection. This methodology involves simulating single event upsets (SEUs) by artificially inducing data errors directly through the software, bypassing the need for physical fault injection hardware. The results of this approach provide invaluable insights into the robustness of the architecture used. The difficulty faced in this work is to inject fault. Injecting errors through software requires an extensive setup of external computers. Our work highlights that the fault injection can be done using the same hardware DUT, making the data acquisition phase of the design much easier.

Other methodologies that provide good insights into how the faults are described in the work done in [6]. Fault injection is approached through simulation-driven techniques, particularly focusing on the interaction and behavior of cores within a locked state. The methodology for injecting faults includes inverting core interface signals during the execution of tests to simulate errors and flipping select state bits within the cores' Register Files (RF) to verify the effectiveness of recovery mechanisms. This changing of signals is done using software and manually. The task of doing this testing manually is daunting. Considering the sheer number of elements in a processor architecture, the time taken to run an iterative test using this methodology puts strain on the development stages of a processor.

The fault injecting methodology used in the work [4] highlights a methodology that is based on hardware-based fault injection. The introduction of the Fault injection module (FIM) that simulates runtime errors by logically OR'ing and AND'ing the output bits of the primary functional unit with the data from Linear Feedback Shift Registers (LFSR) to simulate stuck at 1 and stuck at 0 faults, respectively. This method allows for testing of the processor's fault handling mechanisms by maintaining faults for 3 consecutive cycles to simulate permanent errors and observing if the fault handling logic can isolate the fault unit and switch to the redundant unit for continued operation. The work done in this research paper is analogous to what is done in this work with major differences in how the fault injector module works and the type of faults that it can simulate. Moreover, my work also emphasizes the efficient utilization of hardware resources, incorporating lower LUTs while maintaining robust fault injection capabil-

ities. This optimization enhances the scalability and applicability of the fault injection methodology, ensuring its effectiveness across a wide range of processor architectures and fault scenarios.

The work presented in this thesis shows a direct and practical approach to simulate and test the resilience of processors against space-induced faults. This method allows for a comprehensive assessment of the processor's reliability, offering a novel contribution to the field by bridging the software-induced errors and hardware-induced errors. The ability of the fault injector module designed in this work to integrate seemingly to any processor architecture that has an APB (Advances Peripheral Bus) interface, TCL (Tool Command Language) scripting, and the ability to change the type of fault, duration, and location dynamically during runtime. This provides us great versatility in testing the resilience of the fault.

Chapter 3

Background

3.1 PULP system

Among the available processors, PULP (Parallel Ultra-Low-Power) architecture has the potential to be used in space-based applications because of the inherent design philosophy of parallelization present in the architecture. [6]

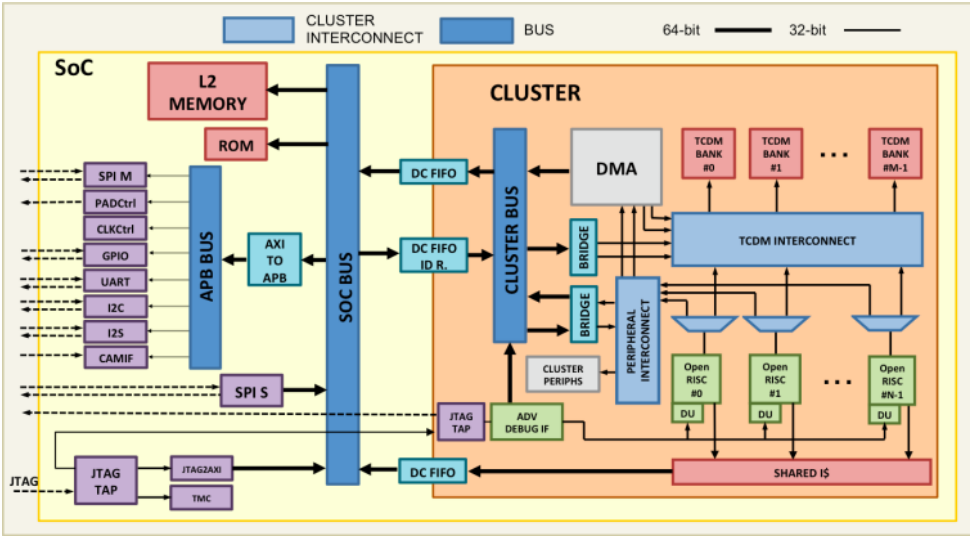

Figure 3.1 – PULP architecture

Fig. 3.1 [1] illustrates the architecture of the PULP. The PULP (Parallel Ultra-Low Power) platform is designed around a cluster of RISC-V cores optimized for energy efficiency and computational performance. It is an open-source platform. This allows for unparalleled flexibility in customizing and adapting the architecture to specific needs, such

1. PULP Platform Repository: `https://github.com/pulp-platform/pulp/blob/master/doc/datasheet.pdf`

as radiation tolerance in space applications.

The PULP platform follows a hierarchical design. The system's design features a fabric controller (FC) core for control, communication, and security functions, along with a cluster of eight cores, each fine-tuned for vectorized and parallelized algorithm execution. This multi-core structure is ideal for implementing fault tolerance mechanisms, such as error detection and correction codes, which are crucial in mitigating single-event faults and stuck-at faults induced by radiation.

The PULP's combination of high-speed shared and instruction memories in the cluster makes it ideal for the execution of code implementing parallel computing. The PULP also features a debug architecture that contains functionalities to help the developer observe/control application code execution.

3.2 Single Event Errors and Stuck-at Faults

The most common types of faults that appear in electronic processors that are exposed to radiation are Single-Event Errors (SEEs) and Stuck-at faults. SEEs are a category of faults that have become increasingly relevant due to the deep scaling in CMOS technology, which, while improving speed and scale of integration, has concurrently highlighted the vulnerability of systems to these faults [3]. SEEs are broadly categorized into two types: non-destructive and destructive.

Non-destructive SEEs, such as Single Event Upset (SEU) and Single Event Transient (SET), induce transient changes in the state of a cell or node without causing permanent damage. These can result in bit flips in digital circuits, which, while not permanently impairing the device's functionality, can lead to erroneous behavior if not properly managed [2].

On the other hand, destructive SEEs, which include Single Event Latch-up (SEL), Single Event Burnout (SEB), and Single Event Hard Error (SHE), lead to permanent damage to the device, potentially causing irreversible failure. The probability of occurrence of these faults increases with harsher environmental conditions, such as those found in

higher radiation belts surrounding Earth, which satellites in geostationary orbit (GEO) or interplanetary missions may encounter [2].

In terms of processors, testing for resistance to SEEs, the fault detection capability is a critical parameter. For instance, in a no-fault condition, the expected normal result (Fn) should match the redundant result (Fr). Any discrepancy indicates a fault within the system [2]. The fault injection method is often employed to simulate SEEs and assess the system's response. This can involve altering data bits to replicate SEU effects or maintaining a fault condition across multiple cycles to simulate permanent errors (PEs). [4].

The design and testing of the fault injector modules, which are essential in emulating the conditions that processors may face in space, focus on simulating SEUs by inverting data bits or generating stuck-at faults by maintaining signal levels opposite at their input state for a duration of time [4]. These modules are integral to validating the fault tolerance capabilities of the processors and often involve extensive simulation campaigns to ensure comprehensive coverage of potential fault scenarios.

3.3 Testbed

The test bed used for evaluating and validating the fault tolerance of the processor designs in this work is the Xilinx Zynq ZCU102 FPGA. ZCU102 is a part of the UltraScale+ family which has a range of integrated features conducive to advanced design testing.

Using this particular FPGA comes with utilizing the Vivado Design Suite, which facilitates a streamlined design process. On top of that, the design files used to program the ZCU102 FPGA, such as constraints, TCL scripts, bitstream, and netlist, are highly modifiable, allowing for rapid iterations and testing of various fault scenarios.

3.4 Programming languages and software

In this section, there is a review of the software and programming languages used in our design.

- **SystemVerilog:** A hardware description and verification language. It was used in the design of all modules comprising the fault injector. The design of the fault injector was verified using the Questasim Advanced simulator and Vivado simulator.
- **TCL script (Tool Command Language):** widely used in Electronic Design Automation (EDA). It was used in our work to automate the modification of the PULP architecture.
- **C-programming language:** used for implementation and giving input parameters for the fault injector to start injecting faults. Using this, it was possible to enable and disable the fault injector module while testing the fault tolerance of the architecture.
- **Git:** used for version control of the design. Enabling to switch between different designs of the fault injector module and also switching designs that target different registers of the PULP architecture.
- **Vivado Design Suite:** used throughout the entire design procedure. Specifically to design the modules, create test benches, check the synthesis design of each module, run TCL scripts, simulation, and verification of the design.
- **Openocd:** Open On-Chip Debugger, used along with GDB (GNU Debugger) for debugging and in-system programming.

Chapter 4

Architecture

This chapter gives a detailed description of the design of the fault injector module and other modules that it comprises. The design of the fault injector is based on the template shown in Fig. 4.1.

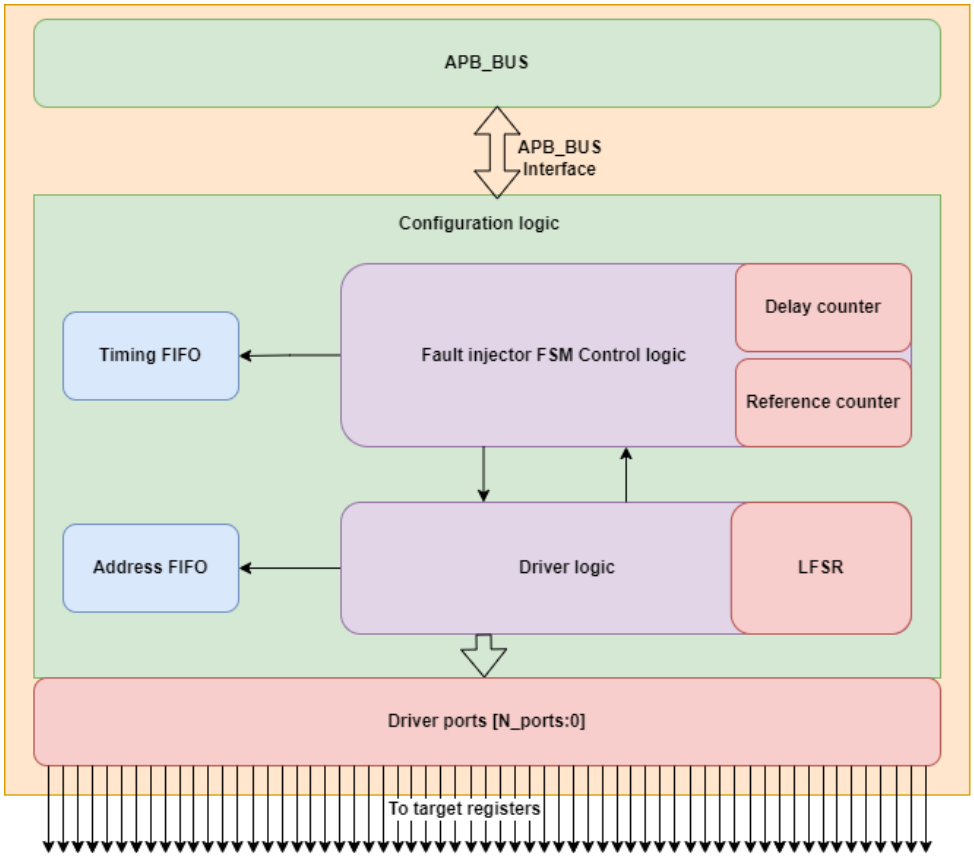

Figure 4.1 – Fault injector architecture

In this template, the module is directly connected to the target registers on the driver

side and connected to the APB BUS on the control side. The fault injector module is integrated as a part of the SOC peripherals along with other peripherals of the PULP architecture like the GPIO, timer, etc.

4.1 Key features of the design

The fault injector module enables precise simulation of the faults within the processor's environment. The module's core capability allows for the configuration of a wide array of fault scenarios and it is particularly adept at mimicking SEEs and Stuck-at faults, which are crucially relevant to the faults experienced because of the harsh conditions of space.

The key feature of the module is its real-time fault injection mechanism. This feature allows designers and testers to introduce faults into the system as it operates, enabling us to perform fault injection tests on different types of computational tasks.

The fault injector module can be scaled to different numbers of registers, creating driver ports on its own to drive every register that needs to be tested. This adaptability ensures that the module remains a versatile tool for fault tolerance testing across various hardware implementations.

By leveraging TCL scripting during the synthesis phase, modifications to PULP architecture are made, ensuring that the fault injector module interacts with the system in a controlled and predictable manner, across different hierarchies of the design.

The fault injector module also provides the location of the faults and the time of the fault it injects. This feature can be utilized to read this information during the operation of the PULP architecture using the APB BUS interface.

The fault injector module incorporates the feature to be configured during its working stages, using the APB bus interface. This enables us to manage the metrics of the faults it injects on the go. It allows for easy control of the delay between the faults, the duration of time it will inject the same fault, and control the start and stop of the

fault injection operation. All this can be achieved using the APB Bus interface anytime during the normal operations of the PULP architecture, providing us with a module that is versatile and easily configurable. Moreover, the fault injector module can be integrated into any scalable processing platform with a peripheral bus with minimum Register Transfer Level(RTL) code changes.

4.2 Design space exploration

The crucial focus of the preliminary phase was to identify a way to modify the PULP architecture that enables us to inject faults in particular targeted registers. In this section, There will be a brief description of the design modification strategies that were explored.

4.2.1 Direct SystemVerilog code Modification

One of the methods discussed was to directly modify the SystemVerilog code of the PULP architecture. The complexity arises from the intricate and interconnected nature of the hardware description languages, where changes can have cascading effects. This necessitated the need for a meticulous verification process for each iteration. The primary concerns with this approach were the potential for introducing errors and the sheer time consumption of the task, rendering it impractical for rapid iteration of the design.

4.2.2 Netlist file modification

The other alternative strategy was to utilize the .edif files that can be generated for the netlist. The idea was to change the .edif files that regulate the rules of the netlist, adding new logic to it and connecting these changes. Though scripting in Python or C to edit the .edif files could theoretically target the necessary registers, this approach lacked the flexibility and efficiency required.

4.2.3 TCL scripting

Using Vivado, it was possible to find a way to modify the design using simple TCL scripts [1]. The TCL scripting feature that Vivado employs to control the design flow provided an efficient way. Its utilities lie in its capability to interact with hardware synthesis tools, enabling automation and precise modifications to the design without needing to manually alter the HDL code. This method greatly facilitated the process of implementing targeted modifications, such as configuring specific registers for fault injection within the PULP system. On top of that, it also provided features to map the connections between the driver ports and the targeted registers into a .csv file which was crucial in determining which drivers will inject faults in which registers.

4.2.4 Conclusion

After exploring the viability of these strategies, it was concluded that the TCL scripting offers superior functionality to our application. Its efficiency, flexibility, and reduction of potential errors compared to other modification strategies that were tested. It enabled automation of the process for implementing changes within the PULP architecture, enabling rapid, iterative changes within the Vivado Design environment itself. The design of the fault injector module is made in a way such that the target registers can be determined during the synthesis phase and connected to the fault injector module. All of this is done by TCL scripting.

4.3 Address Generation

The address to which the fault will be injected is generated as a pseudorandom sequence provided by the LFSR.

4.3.1 Working Principle of LFSR

Linear Feedback Shift Registers (LFSRs) are a type of shift register where the input is a linear function of its previous state [2]. The most common use of LFSRs is in generating pseudo-random sequences based on an initial input. In LFSR, the feedback path is

1. TCL Script documentation: *https://docs.xilinx.com/r/en-US/ug894-vivado-tcl-scripting*
2. LFSR `https://en.wikipedia.org/wiki/Linear-feedback_shift_register`

determined by a polynomial equation. The taps, selected based on this polynomial, are XORed together to produce the new input bit. For example, for an 8-bit LFSR, a typical polynomial might look like this:

$$x^8 + x^6 + x^5 + x^4 + 1$$

This indicates that the 8th, 6th, 5th, and 4th bits are XORed to generate the feedback bit. This mechanism ensures the generation of a maximal-length pseudo-random sequence, provided the polynomial is chosen correctly. The polynomial has to be chosen in a way that ensures maximum periodicity of the LFSR. The periodicity in the context of LFSR refers to the number of cycles an LFSR takes to reach the same sequence as the initial seed value. Once it reaches the initial seed value, the LFSR repeats the sequence. It is to be noted that only a certain type of polynomials allow for maximum periodicity, ensuring that it generates all possible sequences before starting to repeat itself[3]. Fig. 4.2[4] shows a 16-bit Fibonacci LFSR.

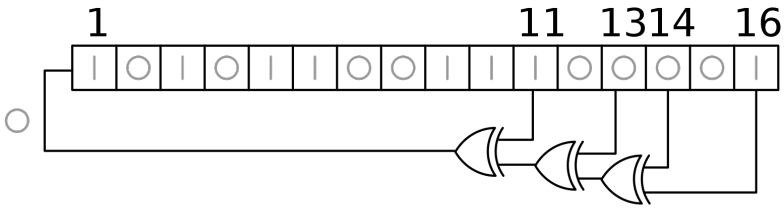

Figure 4.2 – A 16-bit Fibonacci LFSR

4.3.2 Implementaion of the LFSR

The Address generator module in the design reflects this working principle. The address generator generates pseudo-random sequences, which are then utilized as addresses targeting specific registers within the PULP architecture. Each sequence produced corresponds to an address location, enabling a full sweep across various register locations for injecting faults. The implementation begins with initializing the LFSR with a seed value which can be given as an input using the APB BUS interface and this

3. Maximal Length LFSR https://users.ece.cmu.edu/~koopman/lfsr/index.html
4. LFSR https://en.wikipedia.org/wiki/Linear-feedback_shift_register#/media/File:LFSR-F16.svg

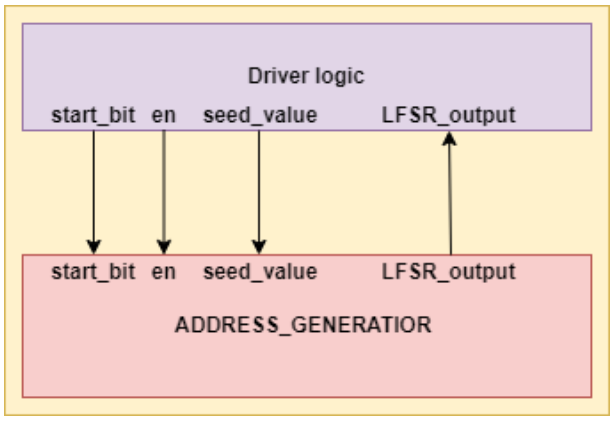

Figure 4.3 – Address module connections

process is controlled by the start bit input of the address generator module. The start bit enables us to take into account a seed value if it is available, or else pick up a default value which in our case is a 32-bit hexadecimal number $32'hDEADBEEF$.

As the LFSR operates when it is enabled, the XOR logic embedded within the module calculates the next bit based on the current state of specific tapped bits, as dictated by the polynomial equation. This bit is then shifted into the register, with the entire register shifting one position, thereby generating a new pseudo-random value every clock cycle. This process ensures a varied and unpredictable pattern of addresses, essential for effective fault injection and testing strategies. The address generator module is controlled by the fault injector's driver logic. Fig. 4.3 shows how the Address module communicates with the driver logic which uses the output of LFSR as addresses to inject fault into the system.

4.4 Counters

4.4.1 Working principle

Counters are fundamental digital devices used in electronics and computing to keep track of occurrences or events by incrementally increasing or decreasing their value in response to input signals. The primary working principle of counters involves sequential state transitions in response to clock pulses, which are represented as a series of flip-flops connected in a specific configuration. In typical scenarios, each pulse on the counter's clock input increments or decrements the count held within the device.

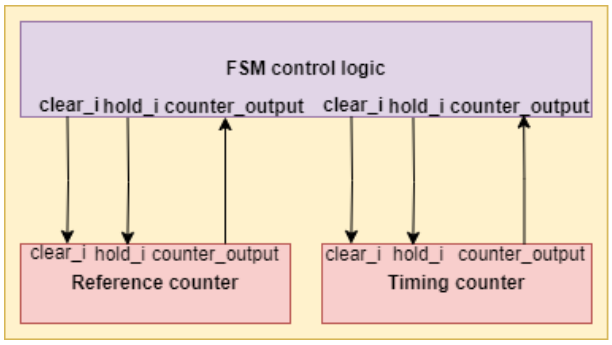

Figure 4.4 – Counter module connections

4.4.2 Implementation of counters

The counter module is implemented twice, each instantiated for different purposes and used by the FSM control logic. The implementation of the counter is straightforward, it increments its value with every positive edge of the clock signal provided it is not held, cleared, or reset. The counter is designed to be 32-bit to ensure it can handle a wide range of counts without overflow, which is particularly useful when dealing with scenarios that may introduce significant delays.

There are two counters in this design; the timing counter is used to keep track of the delay period between faults, and the reference counter is used to keep track of the number of clock cycles that have passed since the start of the operation, which essentially provides the information about the time of the fault injection. Fig 4.4 shows how both counters are connected to FSM Control logic. Keeping the clear input signal high would cause the counter to clear its registers, and initialize to zero. The hold signal, when high will stop the counter from incrementing, essentially holding the previous value until it is pulled low.

4.5 Buffers

4.5.1 Working principle

In our design, FIFO buffers are used extensively to store the fault address and timings of the fault. A FIFO (First In, First out) buffer is a type of data structure used extensively in hardware design and computer science to manage data packets or infor-

mation units in a sequential order. The fundamental principle of a FIFO ensures that the order in which data enters the buffer is the same order in which it exists. This characteristic is why it was picked to be used in the design, as our design has situations where the data timing and order are critical.

There are many ways a FIFO can be implemented, but the working principle is similar. FIFOs operate on two primary pointers: the read pointer and the write pointer. The write pointer tracks where the next data element will be inserted into the buffer, while the read pointer tracks the location of the next element to be read and removed from the buffer.

4.5.2 Implementation

In our design, there are two FIFO buffers. The Address FIFO buffer and the Timing FIFO buffer.

4.5.2.1 Address FIFO

The address FIFO buffer is used to store the address to which the faults are injected. The address FIFO is 32-bit wide, with a buffer depth of 32, both can be changed during the synthesis process based on our need and the number of target registers in our design to inject fault on. Fig. 4.5 shows how the address FIFO buffer is connected. The address buffer is controlled by the control logic. Every time the FSM control logic reaches the state of PULSE, it samples the fault address that is generated by the driver logic by making the input signal valid high. When the configuration logic makes the input signal ready high depending on the data received from the PWDATA of the APB bus interface, it outputs the stored data into the PRDATA of the APB bus. The process of reading data from the address buffer clears that particular part of the buffer. This makes sure the buffer can be emptied by simply reading the data from it, making room for more data to be stored in the buffer.

4.5.2.2 Timing FIFO

The timing FIFO buffer is used to store the timing of the faults injected. Similar to the address buffer, the timer buffer is also 32-bit wide, with a buffer depth of 32. This

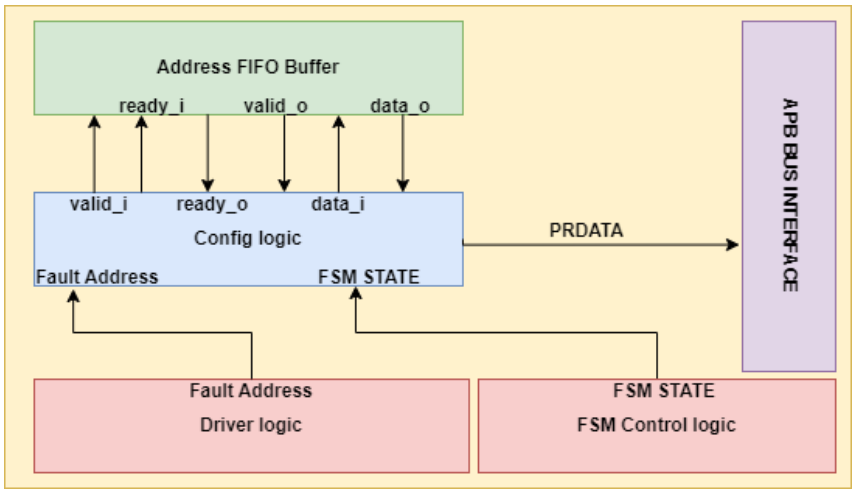

Figure 4.5 – Address FIFO buffer connections

can also be changed during the synthesis process based on our need and the number of target registers in our design to inject fault on. Similar to the address FIFO buffer, the configuration logic also controls the timing buffer, and the data to be stored in the timing buffer comes from the reference counter which is controlled by the FSM control logic. The input signal valid is made high at the same time as the address buffer's input signal and stored at the same time. This ensures the timing of the fault address generation is recorded into the buffer for later use. When the configuration logic makes the input signal ready high, depending on the data received from the PWDATA of the APB bus interface, it outputs the stored data into the PRDATA of the APB bus. Similar to the address buffer, the process of reading data from the timing buffer clears that particular part of the buffer. The data stored in the timing buffer is synced with the address buffer, making sure that if the data is read from the address buffer and timing buffer in the same order (one after the other) gives us the fault address and the timing at which this particular fault address is used by the driver logic. This is all synced with the FSM control logic to ensure when the PULSE state of the FSM control logic is detected, the data is stored in the buffers.

4.6 FSM Control logic

The design and implementation of the fault injector Finite State Machine (FSM) control logic were aimed to be simple and efficient. This module is crafted to orchestrate the timing and execution of the fault injection processes, adhering to predefined opera-

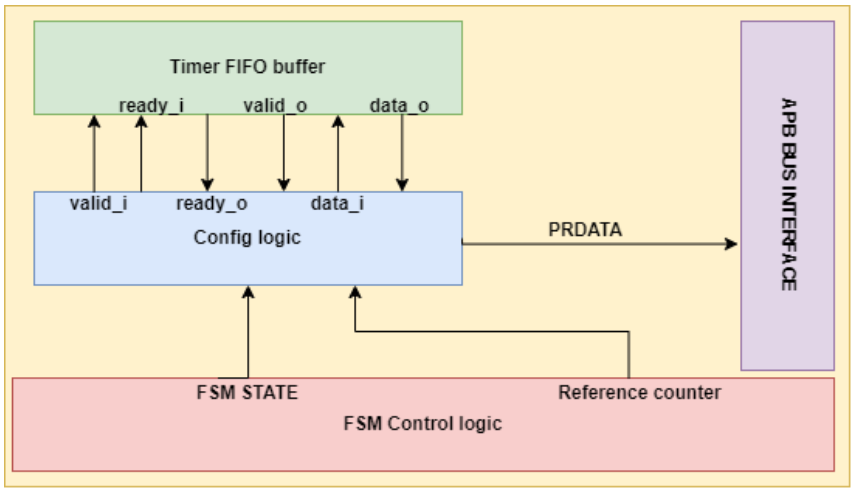

Figure 4.6 – Timing FIFO buffer connections

tional parameters or new operational parameters that are given to it through the APB Bus interface. At the core of the FSM control logic lies its ability to accurately monitor and control the fault injection timeline, from the initiation of the fault injector module to the application of faults within the target system.

4.6.0.1 Operation Dynamics

At the heart of the FSM control logic is a Finite State Machine (FSM) that has three states: IDLE, DELAY, and PULSE. These three states switch within each other once the fault injector module starts its operations. The FSM control logic takes the following inputs from other modules in the fault injector design.

- **Start operation:** This is an input control signal, that is taken from the PWDATA of the APB BUS interface. When this input signal is HIGH, the fault injector module will start its operations. The FSM in this module will not progress unless this input control signal is made HIGH.
- **DELAY CYCLES:** This is a 32-bit wide input data, that determines the periodicity with which the faults will be injected. This input data is taken from the APB BUS interface and is given by the user. If this input data is not given by the user and the user has initiated the operation of the fault injector module, this takes a default value (which can also be changed during the synthesis phase by simply changing it in the params.svh header file).
- **PULSE WIDTH:** This is a 32-bit wide input data, that determines how long the

injected fault will keep flipping the bits of the input to the target registers. This input data is taken from the APB BUS interface and is given by the user. If this input data is not given by the user and the user has initiated the operation of the fault injector module, this takes the default value (which can also be changed during the synthesis phase by simply changing it in the params.svh header file). Keeping this as 1' (one) would essentially simulate SEEs.

The output of the FSM Control logic is the current state of the FSM itself. The FSM control logic does nothing more than to keep the fault injector working in the right state. Its only job is to use the counters and change the states at the right time. The FSM output is a 2-bit state variable, that is taken as an input by all the other modules.

Apart from the counter that is used by the FSM control logic, there is an additional counter called the reference counter which is responsible for keeping track of the clock cycles from the start of the operation. The output of this counter is directly given as an output of the FSM control logic to be used by other modules in the design. This is particularly used by the Configuration logic module that uses this output data to be stored in the timing buffers when the FSM control logic outputs the right state.

4.6.0.2 Finite State Machine

The fig. 4.7 shows the operation of the finite state machine in this module. All the condition checks that are made by the FSM are based on the counter inside the FSM Control logic module. The operations of each state are as follows.

- **IDLE State:** The IDLE state is the default state of the FSM. In this state, the FSM is designed to keep checking the counter value and the input data $DELAY\ CYCLES$ value. It will remain in this state until the start-bit is made high. Once the start-bit is made high, it will check the current counter value (which is incrementing every clock cycle) with the $DELAY\ CYCLES$ input data value. Once the counter value is equal to the $DELAY\ CYCLES$ value, it will transition to the DELAY state, else it will stay in the IDLE state.
- **DELAY State:** Upon entering this state, the FSM checks if the counter value is within the limit $DELAY\ CYCLES + PULSE\ WIDTH$. It will transition to the PULSE State and also stop the counter to ensure that the counter doesn't increment in

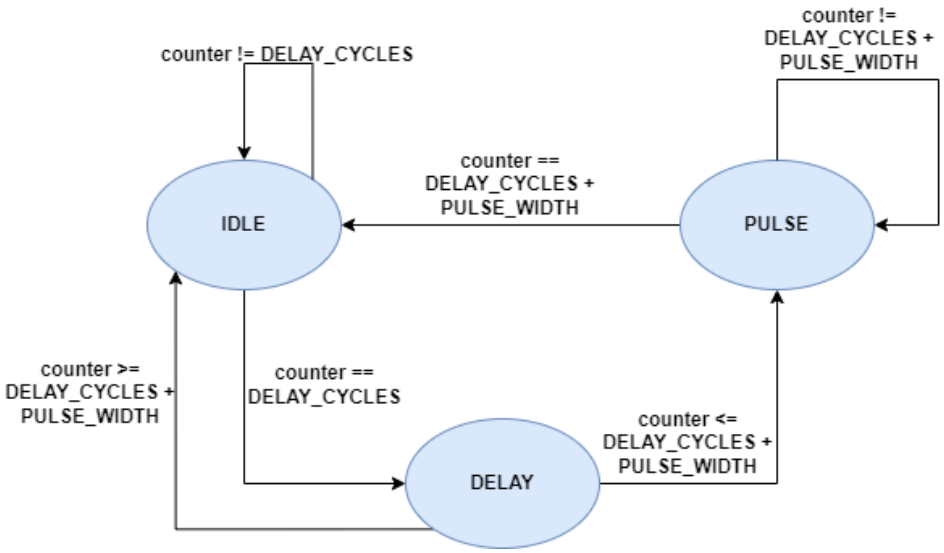

Figure 4.7 – FSM control logic state machine diagram

this DELAY state. If the counter value is not within the limit of $DELAY\ CYCLES + PULSE\ WIDTH$ it will transition back to the IDLE state.

- **PULSE State:** This is the state where the signal to inject fault is issued. If the FSM is found to be in this state, it means that the fault injection is in progress. Once it is in this state, it will check whether the counter value is equal to $DELAY\ CYCLES + PULSE\ WIDTH$, if equal, it will clear the counter and go back to the IDLE state. if not equal, it will increment the counter, and continue to stay in this state. This makes sure that for a longer $PULSE\ WIDTH$ duration, the FSM stays in the PULSE state, ensuring that fault is injected for the required duration.

4.7 Drivers design

The driver module does the crucial job of injecting the faults into the target registers. It relies on the use of LFSR to produce addresses to which the faults will be applied. The operation of the driver module will be discussed below.

4.7.1 Operation of the driver module

The basic connections of the driver module can be seen in Fig. 4.8. The driver module operates based on the state output by the FSM control logic. The number of output ports driven by this module depends on the number of target registers. This is a parameter that must be changed during the synthesis phase of the design. To change the

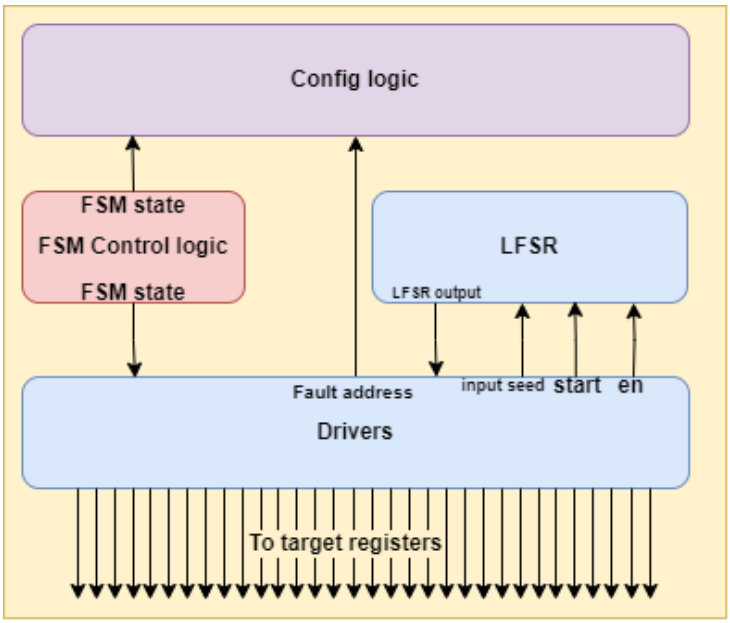

Figure 4.8 – Driver module connections

number of driver ports, one must change the assignment to the variable *N_ports* in the *params.svh* header file of the design.

The driver module uses the output of the FSM control logic to perform its operation. As seen in section 4.6, there are three states, the operation of the driver under each of these states is as follows:

- **IDLE State:** In this state, the driver module drives all the output ports down, making sure that no output port retains its value from the previous fault.
- **DELAY State:** In this state, the crucial step of giving the seed value to the LFSR is done. Based on the output of the LFSR, this state decides whether to use a new seed value received from the user or use the previous output of the LFSR as a new seed value. The idea of using the previous output of the LFSR as a new seed value is done to have the LFSR cycle through all of its available possible sequences given an initial seed value. This is achieved by controlling the start bit and assigning the input of the LFSR to either the seed value or the previous value.
- **PULSE State:** In this state, the fault is applied and the output port that drives the fault is decided by the address that is generated by the LFSR in this state. A new address sequence is generated by the LFSR each time the FSM Control logic

transitions from the DELAY state to the PULSE State and because of the way the FSM control logic is designed, it will always transition from the DELAY state to the PULSE State.

Thus the driver module precisely acts and injects faults at the right time and duration. The output driver ports are connected to the target registers using the TCL scripting. It is completely possible to use TCL commands like *get_cells*, and *get_nets* to get the target registers and connect them to the driver ports. This will be further explained in the upcoming sections. This enables us to connect to any part of the PULP architecture while the fault injector module is still connected as a peripheral.

4.8 Configuration logic

The configuration logic module within the fault injector design is a critical component that organizes various elements, ensuring they operate in sync to achieve desired fault injection outcomes. The module is responsible for the following things.

- **Data storage:** It manages data flow between the driver logic, which generates the fault address, and the FSM logic, which provides the time of the fault. By interfacing with the address and timing FIFO buffers, it makes sure that the data is stored during the PULSE State and can be read using the APB bus interface's *PRDATA* line.
- **API interface:** The module utilizes the APB protocol for configuration settings, receiving operational parameters such as delay cycles, pulse width, and start operation signals. This is achieved by reading the corresponding values from the APB bus's *PWDATA* line.
- **Initiation of the operations:** An important function of this module is to interpret the start operation command, which is a signal received from the APB interface that triggers the commencement of the fault injection process.
- **Seed value configuration:** It also reads the seed value from *PWDATA*, which is used to initialize the LFSR in the driver logic, influencing the sequence of the fault address.
- **Address decoding logic:** This module does the essential task of parsing the incoming addresses on the *PADDR* line, enabling appropriate read or write opera-

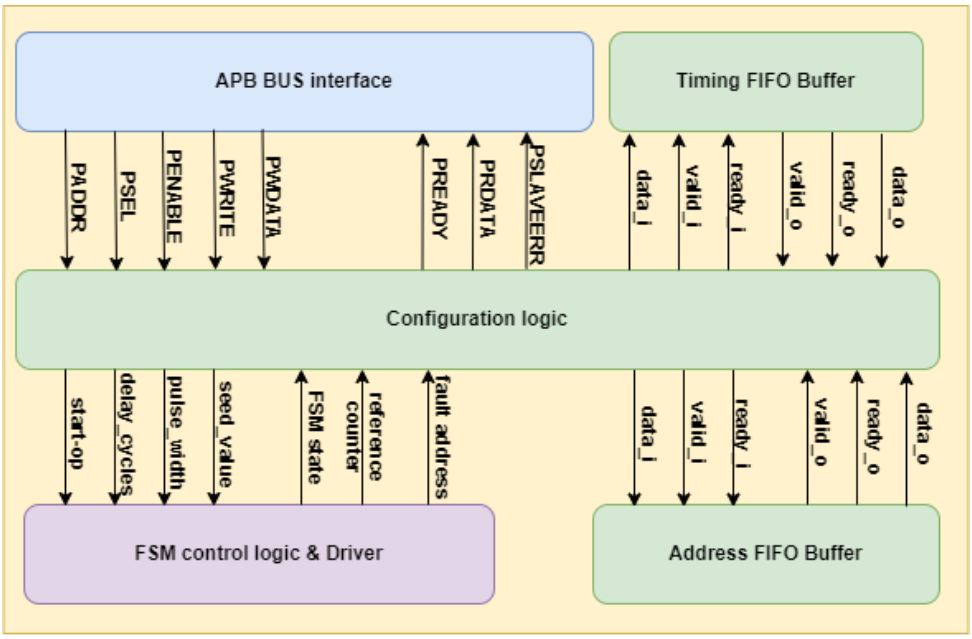

Figure 4.9 – Configuration logic module connections

tions depending on the address specified.

- **Error management:** The module ensures system integrity by driving the *PSLAVEERR* signal when erroneous conditions are detected. The erroneous condition in terms of the fault injector would be the module reading the wrong address from the *PADDR* line.

4.8.1 Operation

The configuration logic module acts as a central hub, decoding the APB bus addresses to trigger specific actions within the fault injector. It maintains the synchronization between the storage of the fault address and timing parameters, only allowing data storage during the PULSE State to avoid duplication. The read-and-write logic is constructed to handle data exchanges with the APB bus, ensuring that configuration data is received properly. It interfaces directly with the APB bus to manage and configure the fault injection parameters. The module harnesses the APB bus signals such as *PADDR* for address decoding, *PWRITE*, and *PENABLE* for determining operation modes, and *PWDATA* for configuring the operational parameters. The *PREADY*, *PRDATA*, *PSLAVEERR* are used for managing the readiness of the data transfer and error signaling.

This module controls all the other modules in the design, making sure they are all synced to the FSM control logic's state output. Fig. 4.9 shows how the inputs and outputs to the other modules are governed by this module.

4.8.2 Address allocations

This subsection discusses how the configuration logic module interprets and responds to address signals from the APB bus to manage the state and behavior of the fault injector design. The *PADDR* line of the APB bus is 32-bit wide. For the design of the fault injector module, the whole 32-bit of the signal *PADDR* is not utilized, instead, only the last 16 bits are utilized by the decoding logic. Table 4.1 shows the address mapping used by the fault injector module.

Address	**Operation**
$0x0000$	Start operation
$0x0001$	Delay cycles configuration
$0x0002$	Pulse Width configuration
$0x0003$	Seed value configuration
$0x0004$	Address buffer read operation
$0x0005$	Timing buffer read operation

Table 4.1 – Fault Injector Module Address Mapping

- **Start operation:** The address $0x0000$ is allocated for initiating the fault injection process. Writing to this address triggers the beginning of operations by setting the fault injector into an active state.
- **Delay cycles configuration:** The address $0x0001$ controls the delay cycles. Data written to this address sets a delay period between each fault occurrence.
- **Pulse Width configuration:** Address $0x0002$ is designated for adjusting the pulse width, which dictates the duration of fault once injected. This is the parameter that is responsible for simulating Single Event Errors (SEEs) or Stuck-at faults.
- **Seed value configuration:** The seed value, used to generate the pseudo-random sequence for the fault addresses is configured through address $0x0003$. Modifying this value changes the pattern of fault address generation.
- **Address buffer read operation:** Address $0x0004$ is used for reading from the address FIFO buffer, enabling the system to retrieve stored fault addresses for inspection or further processing.

- **Timing buffer configuration:** Similarly, address $0x0005$ is assigned to read operations from the timing FIFO buffer, which holds the timing information associated with each fault.

Each address acts as a specific register within the configuration logic, and writing or reading from these addresses allows the system to configure and monitor the fault injection process dynamically. This structure ensures a modular and easily extensible system where modifications can be made through simple APB transactions without the need for direct hardware intervention.

4.9 APB bus Design

The fault injector module communicates with the Fabric Controller (FC) using the APB Bus. Since the fault injector module is connected as a peripheral, the fault injector module can work independently from the FC and inject faults.

4.9.1 Protocol description

The Advanced Peripheral Bus (APB) protocol is part of the AMBA (Advanced Microcontroller Bus Architecture) suite of protocols. It is designed for low-bandwidth control accesses, for example, peripheral or control registers. Below is a table (Table 4.1) [5] of the APB protocol signals:

Signal	**Source**	**Width**
PCLK	Clock	1
PRESETn	System bus reset	1
PADDR	Requester	ADDR_WIDTH
PSELx	Requester	1
PENABLE	Requester	1
PWRITE	Requester	1
PWDATA	Requester	DATA_WIDTH
PREADY	Completer	1
PRDATA	Completer	DATA_WIDTH
PSLVERR	Completer	1

Table 4.2 – APB Bus Protocol Control Signals

The signals described in the table 4.1 are used as follows:

5. APB bus protocol `https://developer.arm.com/documentation/ihi0024/latest/`

- **PCLK:** This is the clock signal for the APB protocol. All transactions on the APB are synchronized to the rising edge of PCLK.
- **PRESETn:** An active-low signal that resets the APB interface logic.
- **PADDR:** This bus carries the address code to the peripherals. The width can vary to accommodate the system address space. In the PULP architecture, this is fixed to 32-bit wide.
- **PSELx:** This signal is asserted by the master to select the corresponding slave device.
- **PENABLE:** A signal to indicate that the current transfer is active.
- **PWRITE:** This signal indicates the direction of the data transfer; if high, it's a write operation, and if low, it's a read operation.
- **PWDATA:** The data bus carrying information from the master to the selected slave during a write transaction. In the PULP architecture, this signal is also 32-bit wide.
- **PREADY:** A signal from the slave to the master to indicate the end of the transfer cycle.
- **PRDATA:** The data bus carrying information from the slave to the master during a read transaction.
- **PSLVERR:** An optional signal that indicates an error in the transfer.

4.9.2 Implementation of the protocol

The entire fault injector module is integrated into the PULP architecture using a wrapper module. The wrapper module takes care of parsing the APB bus slave interface into their appropriate signals and giving it as input to the fault injector module. The integration of the APB bus into the fault injector is made easy because of the way the configuration logic is designed. It is designed to directly take the inputs and outputs of the APB bus slave interface and use it for its operation. Fig. 4.10 shows the wrapper module that takes in the APB bus slave interface and assigns the input and outputs of the interface to the fault injector module, effectively enabling communication with the processor.

```
35
36  module fg_apb_m3_wrap(
37      input logic clk_i,
38      input logic rst_ni,
39
40      APB_BUS.slave apb_slave,
41
42      output logic [N_PORTS-1:0] fg_output_driver
43  );
44
45  fg_fifo_config fault_injector (
46    .fg_fifo_config_clk_i(clk_i),
47    .fg_fifo_config_rst_ni (rst_ni),
48
49    .PADDR(apb_slave.paddr),
50    .PSEL(apb_slave.psel),
51    .PENABLE  ( apb_slave.penable ),
52    .PWRITE   ( apb_slave.pwrite  ),
53    .PWDATA   ( apb_slave.pwdata  ),
54    .PREADY   ( apb_slave.pready  ),
55    .PRDATA   ( apb_slave.prdata  ),
56    .PSLAVEERR( apb_slave.pslverr ),
57
58    .fg_fifo_config_driver_ports(fg_output_driver)
59  );
60
61  endmodule
```

Figure 4.10 – APB bus wrapper module

The fault injector module is allocated a unique address space within the PULP architecture to facilitate clear and efficient communication over the APB bus. The defined starting address for the fault injector module is $32'h1A12\_0000$, and the ending address is $32'h1A12\_FFFF$. This dedicated address range ensures that the fault injector can be accessed and managed without interference from other peripherals, providing a clean and isolated namespace for control operations.

4.9.2.1 Address Space Utilization

- **Start Address:** $32'h1A12\_0000$ Marks the beginning of the fault injector's command space. Specific offsets from this base address are used to access various control registers within the fault injector module.
- **End Address:** $32'h1A12\_FFFF$ Signifies the end of the fault injector's addressable space, providing ample room for future expansion of control registers or additional functionalities. The allocation of a contiguous address space simplifies the design and enhances the modularity of the PULP system, allowing for straightforward mapping of the APB bus signals to the fault injector's inputs and outputs.

4.10 Configurable Parameter Pre-synthesis:

The following parameters can be changed before the synthesis. The header file used by the fault injector module (params.svh) has initialization values for the following and each can be modified as per requirements.

- *N_ports:* This parameter must be equal to the number of target registers. This dictates the number of output ports that needs to instantiated by the fault injector module.
- *BUFFER_DEPTH:* This parameter dicates the size of address and timing FIFO buffers.
- *DEFAULT_DELAY_CYCLES:* This parameter can be used to set the default delay between faults, this value will be used if *Delay_cycles* is not provided through APB bus interface.
- *DEFAULT_PULSE_WIDTH:* This parameter is used to provide the default fault duration. This will not be used if *PULSE_CYCLE* is fed into the fault injector module using APB bus interface.
- *DEFAULT_SEED_VALUE:* This value dictates the initial value given to the LFSR for pseudo-random sequence generation. This can be given as an input through APB bus interface else, it will use the default specified value.

4.11 Modification of the Processor Architecture

To inject fault into the target registers in a processor architecture, it becomes necessary to modify some elements of architecture. As discussed in section 4.2, the conclusion led to the development of a custom TCL script. This script automates the process of introducing new hardware elements around target registers, facilitating the injection of faults. The number of target registers is first determined using TCL scripts and Vivado Design Suite. Using this value of number of registers, the params.svh file is changed to accommodate the number of output drivers needed to connect to all the required target registers and then the modified design is synthesized. Once the synthesis is done, the custom script that was developed for modification of the processor architecture is used either using the TCL console or TCL tool in Vivado Design Suite.

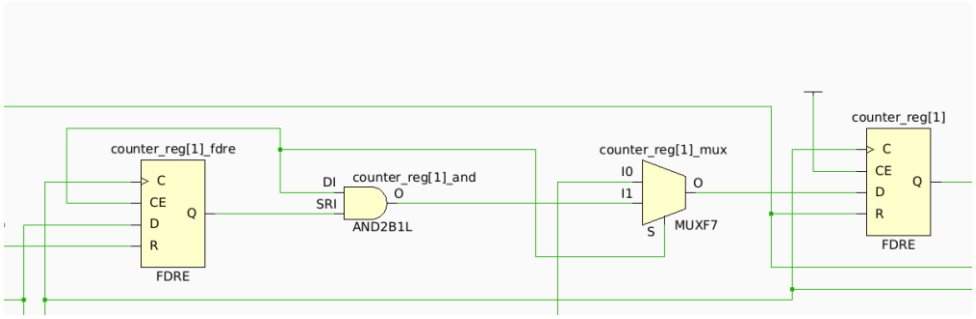

Figure 4.11 – Modified target register in PULP architecture

4.11.1 New Design Elements

Fig. 4.11 shows a single targeted register modified using the TCL script. To enable precise control over fault injection within the architecture, the implementation required the introduction of specific logic elements. These elements were inserted near the target registers and manipulated through a TCL script to modify the pre-existing netlist dynamically. These elements, sourced from the Xilinx Vivado Design Suite libraries [6], include AND2B1L, FDRE, and MUXF7, which play crucial roles in the fault injection process.

4.11.1.1 AND2B1L

The AND2B1L is a two-input AND gate utilized to replace a Configurable Logic Block (CLB) latch. It's adept at reducing logic levels and enhancing logic density, optimizing the trade-off between register/latch resources and logic utilization. As shown in Table 4.2, the operation of this logic block is not that of a true AND gate. The input DI is connected to the fault injector output driver and the SRI to the output of the FDRE block which mimics the input data line of the original register. The output (O) of this logic block is connected to the I1 input of the MUXF7 block.

DI	SRI	O
0	0	0
0	1	0
1	0	1
1	1	0

Table 4.3 – AND2B1L Truth Table

6. Vivado Design library elements `https://docs.xilinx.com/r/en-US/ug974-vivado-ultrascale-libraries/Design-Elements`

4.11.1.2 FDRE

The FDRE is a D Flip-Flop with Clock Enable and Synchronous Reset, forming the core of the newly integrated fault injector logic. It ensures that only when the Clock Enable (CE) is active, and the reset (R) is not asserted, the data is transferred to the output (Q) in synchrony with the clock. This element's introduction is pivotal for maintaining synchronization and ensuring that faults can be injected and retracted in sync with the system's clock. Table 4.3 shows the truth table of the FDRE. The input of the FDRE is connected to the data line of the target register and the output is connected to the SRI input of the AND2B1L logic block. The Clock and reset pins of the FDRE are connected to the system clock and reset while the CE pin is connected to the fault injector's output driver port. This means that the FDRE Flip-Flop is enabled to sample just when the fault is injected and once the signal from the fault injector module's output driver port injects the fault and stops, the CE pin is pulled low, making the FDRE hold any input it sampled, enabling for more versatility with the design.

R	CE	D	Q (next state)
1	X	X	0
0	0	X	No Change
0	1	D	D

Table 4.4 – FDRE Truth Table

4.11.1.3 MUXF7

The MUXF7 is a multiplexer that assists in creating complex logic functions within a single CLB. In the fault injection context, the MUXF7 is paramount in selecting between the original data path and the fault-injected path, effectively controlling when and where faults are introduced into the system. Table 4.4 shows the truth table of the MUXF7 block. MUXF7 is a true multiplexer, that allows selection of 1 of the two input signals. The I0 input of the MUXF7 block is connected to the original data line of the target register while the I1 input of the MUXF7 block is connected to the output of the AND2B1L. The select input (S) is connected to the fault injector's output driver ports.

Each element was carefully chosen for its specific properties that contribute to the fault injector's functionality—such as the AND2B1L's ability to flip logic state under fault conditions, the FDRE's precise control over data synchronization, and the MUXF7's

S	I0/I1	O
0	I0	I0
1	I1	I1
X	0	0
X	1	1

Table 4.5 – MUXF7 Truth Table

A

```
current_instance i_pulp/cluster_domain_i
/cluster_i/CORE[0].core_region_i
/CL_CORE.RISCV_CORE/if_stage_i/
prefetch_32.prefetch_buffer_i/fifo_i

%get the cell names of these registers
set reg_list [lindex [all_ffs]]

%get the nets that are connected to
%the inputs of these flipflops that we are modifying
set input_nets
[get_nets -of_objects
[get_pins -of_objects $reg_list -filter
{REF_PIN_NAME=~D}]]

%move back to the whole design
current_instance
```

B

```
%create new design elements, nets and name them
%as <original_flipflop_name>_fdre
for {set x 0} {$x< $reg_count} {incr x} {

   create_cell -reference FDRE [lindex $reg_list $x]_fdre
   create_cell -reference MUXF7 [lindex $reg_list $x]_mux
   create_cell -reference AND2B1L [lindex $reg_list $x]_and
   create_net [lindex $reg_list $x]_mux_sel [lindex $reg_list $x]_muxO_Din
   [lindex $reg_list $x]_FDREQ_AND [lindex $reg_list $x]_ANDO_MUXI1
}
```

Figure 4.12 – TCL Script for identifying the target registers and creation of New Design Elements

capability to route the fault signals to the correct locations. The combined operation of all these elements ensures that the input bit to the target register is flipped and successfully enables us to inject a fault into the target registers.

4.11.2 Working of the TCL script

The script's operations are as follows:

- **Identification of Target registers:** The script begins by gathering the registers with the given design. This design can be changed depending on which part of the PULP architecture our target registers reside. Fig. 4.12-A shows the TCL script that gets into the hierarchy of the design in which the target registers are situated, finds the number of target registers, and stores them before switching back to the instance of the PULP.
- **Creation of Additional Hardware Elements:** For each identified register, the script dynamically creates a series of new hardware elements: FDRE(Flip-Flop with Synchronous Reset and Clock Enable), MUXF7 (Multiplexer), and AND2B1L

A

```
%connect the nets the respective elements
for {set x 0} {$x< $reg_count} {incr x} {
connect_net -hierarchical -net
   [get_nets [lindex $reg_list $x]_FDREQ_AND]
-objects  [list
[get_pins -of_objects [get_cells [lindex $reg_list $x]_and]
-filter {REF_PIN_NAME =~SRI}]
[get_pins -of_objects [get_cells
[lindex $reg_list $x]_fdre]
-filter {REF_PIN_NAME =~Q}]]

connect_net -hierarchical -net [get_nets [lindex $reg_list $x]_ANDO_MUXI1]
-objects  [list [get_pins -of_objects [get_cells [lindex $reg_list $x]_and]
-filter {REF_PIN_NAME =~O}] [get_pins -of_objects
[get_cells [lindex $reg_list $x]_mux]
-filter {REF_PIN_NAME =~I1}]]

connect_net -hierarchical -net [get_nets [lindex $input_nets $x]]
-objects [list [get_pins -of_objects [get_cells [lindex $reg_list $x]_mux]
-filter {REF_PIN_NAME=~I0}] [get_pins -of_objects [get_cells
[lindex $reg_list $x]_fdre]
-filter {REF_PIN_NAME =~D}]]

connect_net -net [get_nets [lindex $reg_list $x]_muxO_Din] -objects
[list [get_pins -of_objects [lindex $reg_list $x] -filter
{REF_PIN_NAME=~D}] [get_pins -of_objects
[get_cells [lindex $reg_list $x]_mux] -filter
{REF_PIN_NAME=~O}]]


}
```

B

```
%disconnect the data net from the input and connect to the MUXF7

for {set x 0} {$x< $reg_count} {incr x} {
disconnect_net -net
   [get_nets [lindex $input_nets $x]]
-objects
[list [get_pins -of_objects
[lindex $reg_list $x] -filter {REF_PIN_NAME=~D}] ]

connect_net -net [get_nets [lindex $reg_list $x]_muxO_Din]
-objects [list [get_pins -of_objects [lindex $reg_list $x]
-filter {REF_PIN_NAME=~D}] [get_pins -of_objects
[get_cells [lindex $reg_list $x]_mux] -filter {REF_PIN_NAME=~O}]]
}
```

C

```
%get the output bus pins of our fault injector design
set fault_injector_pin_list
[lindex [get_pins -of_objects
[get_cells -hierarchical  fault_injector_apb]
-filter {BUS_NAME=~fg_output_driver}]]

%connect one by one the mux_sel pin and the  fault injector's output bus pins.
for {set x 0} {$x< $reg_count} {incr x} {connect_net -hierarchical
-net [get_nets [lindex $reg_list $x]_mux_sel]
-objects [list [lindex $fault_injector_pin_list $x]]
-verbose}
```

Figure 4.13 – TCL Script for connection, disconnection and integration

(AND gate with one inverted input). These elements are named systematically based on the original registers, ensuring a clear relationship between the original and the new elements. Fig. 4.12-b shows the TCL script that executes this task.

- **Wiring of New Elements:** The script proceeds to wire the new elements together and to the original design. It carefully constructs nets to connect the multiplexer's outputs to the inputs of the target registers, ensuring the fault signals can be appropriately directed through the new hardware. Fig. 4.13-A demonstrates the TCL script that connects the nets to the inputs and outputs of the new design element.
- **Decoupling Original Connections:** It is crucial to decouple the original flip-flop input connections to prevent interference with normal operation. The script shown in Fig. 4.13-B disconnects the existing nets and reroutes them through the newly inserted hardware.
- **Clock and Reset Integration:** The newly inserted FDREs are connected to the existing clock and reset lines to maintain synchronization with the overall design's timing.
- **Fault Signal Routing:** Finally, the script routes the fault signals from the fault injector output to the select lines of the multiplexers. This step is pivotal as it allows the fault injector to control which registers receive the fault condition,

thereby enabling selective fault injection. Fig. 4.13-C shows a snippet of the TCL script that is responsible for this task.

This automated process not only enhances the efficiency of the modification procedure but makes the iterative changes much faster. The script's flexibility allows for easy adjustments to be made, accommodating changes in the design or the fault injection requirements.

Chapter 5

Design Integration with a Processor Architecture

5.1 PULP SoC Integration

The fault injector module is designed to be integrated into any processor architecture that supports an APB bus interface. The PULP architecture is used as test architecture. The fault injector module is integrated from two ends, one from the input side of the fault injector and the other is the output ports of the fault injector that are connected to the target registers. The APB (Advanced Peripheral Bus) serves as the communication backbone for interfacing the fault injector module within the PULP architecture. This process was particularly made easier because of the already existing APB infrastructure, which the PULP architecture uses to communicate with its other peripheral devices like the GPIOs, Timers, etc. This largely reduced the time to integrate and didn't require extensive modification to the RTL design files of the PULP architecture. The fault injector is assigned a dedicated address space within the PULP memory map, allowing it to operate as a standalone peripheral, akin to existing peripheral components. Fig. 5.1 shows an overview of the integration of the fault injector module into the PULP architecture. It is to be noted that the output port of the fault injector doesn't lead to any external pads like some of the other peripherals in the architecture but instead is connected to the target registers using TCL scripting.

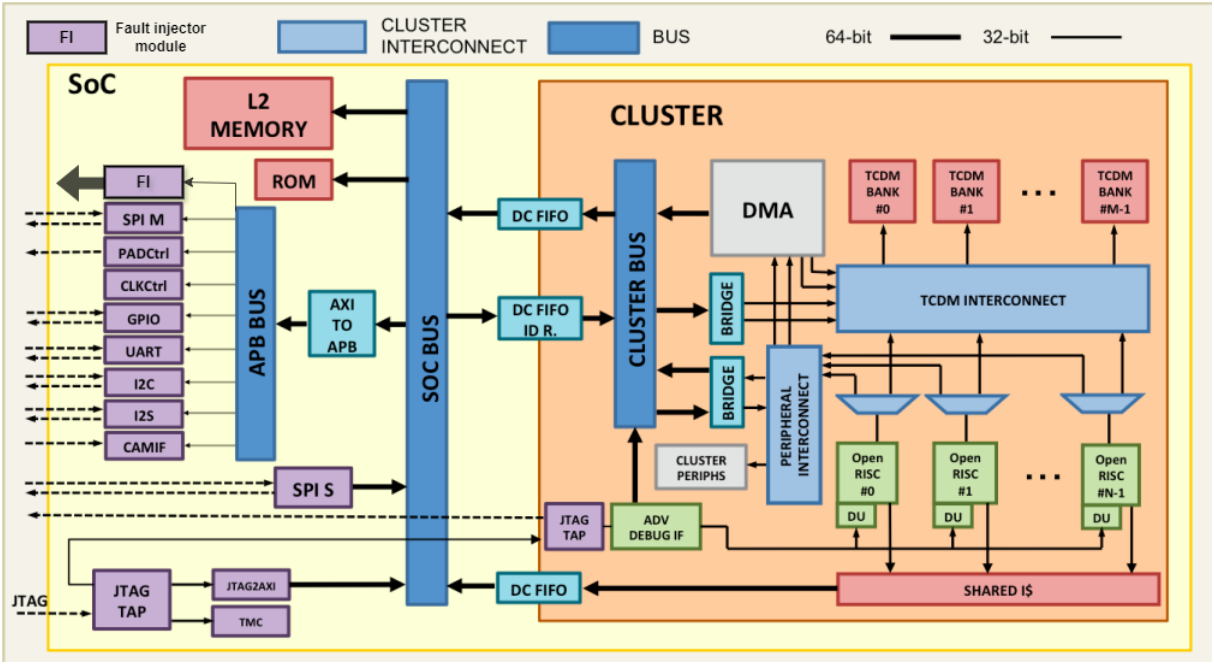

Figure 5.1 – Fault injector integrated PULP overview

The first part of the integration process includes declaring the fault injector wrap module in the SoC peripheral part of the PULP architecture. This can be verified in the Vivado Design Suite, by looking at the SoC peripheral part of the PULP architecture and seeing the fault injector module appear in the hierarchy of the PULP Architecture. Fig 5.2 shows the fault injector module instantiated inside the SoC peripherals module of the PULP Architecture. The second part of the integration process includes the declaration of the APB bus interface using which the fault injector module will communicate with the PULP Architecture. This was done by simply instantiating a new bus with the help of an already existing APB bus interface and using this bus interface as input to the fault injector module. Fig. 5.2 shows the instantiation of the APB bus interface used by the fault injector along with the other APB buses in the SoC peripherals. Some further modifications were done to add additional parameters like *N_PORTS* which are used by the fault injector module to determine the number of output ports in the design.

Table 5.1 shows how the address is allocated to the Fault injector module compared to the order peripherals in the architecture.

```
APB_BUS s_fll_bus ();

  APB_BUS s_gpio_bus ();
  APB_BUS s_udma_bus ();
  APB_BUS s_soc_ctrl_bus ();
  APB_BUS s_adv_timer_bus ();
  APB_BUS s_soc_evnt_gen_bus ();
  APB_BUS s_stdout_bus ();
  APB_BUS s_apb_timer_bus ();


  APB_BUS s_fault_injector_bus (); //fault injector bus
```

```
//fault injector wrap function

logic [N_PORTS-1:0] fg_output_driver;

 fg_apb_m3_wrap fault_injector_apb(
  .clk_i (clk_i),
  .rst_ni (rst_ni),

  .apb_slave(s_fault_injector_bus), //APB_BUS interface as input

  .fg_output_driver(fg_output_driver)
);
```

Figure 5.2 – Fault injector and fault injector APB bus instantiation in the SoC peripherals

5.2 TCL scripted connections

As discussed in Chapter 4, section 4.10, TCL scripting was employed to automate the tedious and error-prone task of manually connecting the fault injector's outputs to the appropriate target registers. This automation process involves several steps:

- **Identification of Target Registers:** The script initiates by scanning the PULP architecture to identify the registers within the specified design level of the PULP architecture. Once the target registers have been identified, the connections of the new elements and their interconnections are added to the design again using TCL scripting.
- **Output to Register Mapping:** For each identified register, the script dynamically generates connections from the fault injector's outputs. This mapping ensures that each register can be selectively subjected to fault conditions during the simulation or testing phases. The script connects the output ports of the fault injector module, to the select pin of the MUXF7 block, to the CE pin of the FDRE block, and to the DI pin of the AND2B1L block.
- **Hierarchical Integration:** Given the complex hierarchy of the PULP SoC design, the script intelligently navigates through various levels of the design, establish-

Table 5.1 – PULP SoC Peripheral Address Allocations

Peripheral	Address Range	Description
FLL	0x1A10_0000 0x1A10_0FFF	Frequency Locked Loop (FLL) for dynamic clock frequency management.
GPIO	0x1A10_1000 0x1A10_1FFF	General-Purpose Input/Output (GPIO) for interfacing with external devices.
SPI Master	0x1A10_2000 0x1A10_3FFF	Serial Peripheral Interface (SPI) Master for serial communication with peripherals.
SoC Control	0x1A10_4000 0x1A10_4FFF	Control unit for managing SoC-wide settings and configurations.
Adv Timer	0x1A10_5000 0x1A10_5FFF	Advanced Timer unit for high-resolution timing and event management.
SoC Event Gen	0x1A10_6000 0x1A10_6FFF	Event Generator for orchestrating event-driven actions within the SoC.
EU	0x1A10_9000 0x1A10_AFFF	Event Unit for managing internal and external event triggers.
Timer	0x1A10_B000 0x1A10_BFFF	Basic Timer unit for general timing and delay functions.
HWPE	0x1A10_C000 0x1A10_CFFF	Hardware Processing Engine for accelerating specific computational tasks.
Stdout	0x1A10_F000 0x1A10_FFFF	Standard Output interface for debugging and printing messages.
Debug	0x1A11_0000 0x1A11_FFFF	Debugging module for system inspection and troubleshooting.
Fault Injector	0x1A12_0000 0x1A12_FFFF	Custom module for injecting faults into the system for testing the resilience and fault tolerance.

ing the connections. This involves traversing modules, submodules, and their interfaces, ensuring that the fault injector's output ports are correctly linked irrespective of the design layer.

An essential part of the integration process is documentation, for which the TCL script automatically generates a CSV file, detailing the connections established between the fault injector's output ports and the target registers. As can seen from Fig. 5.3, the CSV file gives a detailed idea of which output port is connected to which target registers. The traceability is further improved when the fault address that is outputted by the fault injector module through the APB bus corresponds directly to an output port. For example, in Fig 5.3, the output port *fg_output_driver[100]* corresponds to the fault

	A	B	C
1	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[0]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][0]]
2			
3	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[100]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][10]]
4			
5	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[101]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][11]]
6			
7	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[102]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][12]]
8			
9	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[103]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][13]]
10			
11	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[104]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][14]]
12			
13	i_pulp/soc_domain_i/pulp_soc_i/soc_peripherals_i/ fault_injector_apb/fg_output_driver[105]		i_pulp/soc_domain_i/pulp_soc_i/fc_subsystem_i/FC_CORE.IFC_CORE/id_stage_i/registers_i/ riscv_register_file_i/rf_gen[10].mem_reg[10][15]]

Figure 5.3 – Output port mapping of the fault injector

address that is equal to the hexadecimal value of decimal number 100 (which is 0x64). This plays a crucial role in understanding the impact of different faults injected during simulations and testing. The fault injector module is integrated using TCL scripts in such a way that it doesn't significantly affect the timing constraints of the architecture. Fig. 5.4 shows the schematics of connections that are done through the hierarchy of the design. The register on the right-hand side is the target register and the elements on the left-hand side are the interconnected new design elements with the net connecting this design around the target register to the output port of the fault injector.

5.3 Working of fault injector post integration

Once the fault injector module is integrated into the design, it operates under a defined set of procedures that dictate how faults are introduced into the system.

- **Initialization:** Upon global system reset, the fault injector module initializes, sets up the internal registers, and waits for activation commands from the APB interface.
- **Configuration:** Through the APB bus, configuration parameters such as delay cycles, pulse width, and initial seed value are specified. This step involves writing to a specific address mapped to the Fault injector's control registers.
- **Activation:** The module is activated via the APB bus interface. If the configuration parameters are not specified, the fault injector will assume default predefined values and proceed with injecting the faults. These default parameters are predefined in the header files of the fault injector's design.

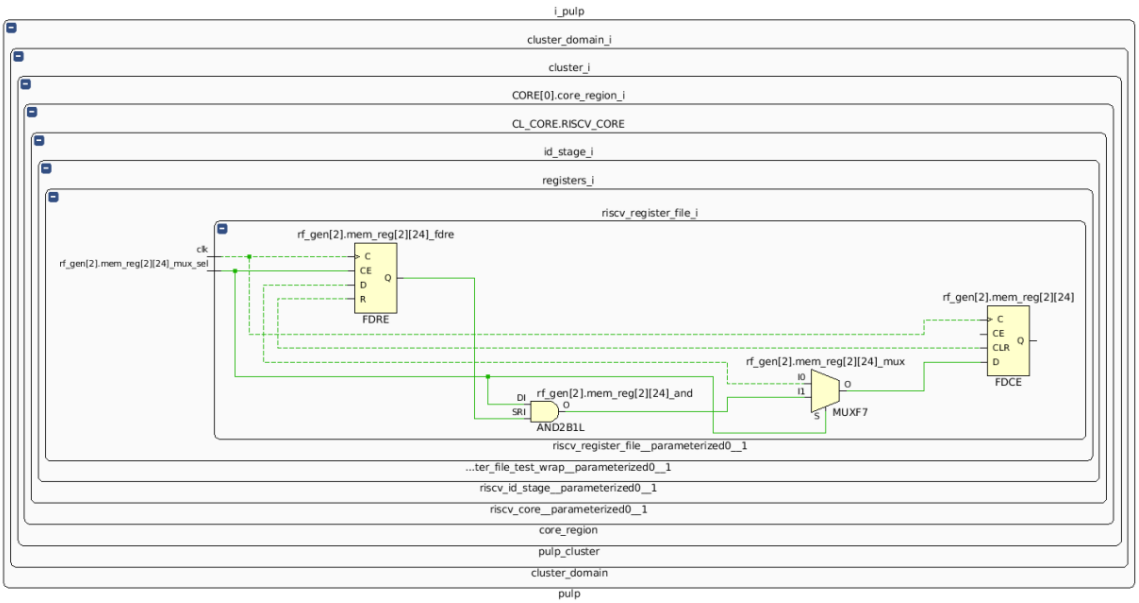

Figure 5.4 – Schematics of the TCL scripted connections

- **Fault injection:** The module alters the logic of target registers, simulating the desired fault conditions. This manipulation is achieved by routing the outputs of the fault injector through the newly added design elements.
- **Monitoring and Feedback:** Throughout the fault injection process, the module stores the time of the fault and the fault address in its buffers. This data can be accessed through the APB interface for analysis and further testing.
- **Termination:** The fault injection cycle can be terminated through the APB bus by making the start bit low.

The integration and operation of the fault injector module within the PULP architecture represent a significant advancement in our ability to rigorously test and enhance the reliability of systems exposed to fault conditions. Through automated scripting and detailed procedural integration, this module not only simplifies the process of fault injection but also lays the groundwork for further innovations in fault tolerance and system resilience. This work underscores the critical importance of such tools in the development and verification of robust electronic systems, promising a future where reliability is as dynamically adaptable as the systems it seeks to protect.

Chapter 6

Validations and Results

6.1 Validation of the design

The validation phase of the fault injector module plays a crucial role in ensuring its reliability and effectiveness within the PULP architecture. This comprehensive testing strategy is designed to cover every component and functionality of the module, ensuring that it performs as expected under various conditions. Each component of the fault injector module underwent rigorous testing through carefully designed test benches that were written in SystemVerilog. These test benches were crafted to encompass a wide range of test cases that mimic operational scenarios the component would encounter during actual use. By adopting this approach, we ensured that every module component adhered to its specified operational and logical functionality.

- For every design phase, a corresponding test bench was developed. This approach allowed for immediate and targeted validation of new or modified components, facilitating early detection and correction of errors.
- The validation process employed both behavioral simulations and post-synthesis functional simulations, utilizing industry-standard tools such as Vivado and QuestaSim. This dual-simulation approach provided a robust validation framework, enabling the detection of issues that may not be apparent in one environment alone.
- The integrated system was subjected to extensive testing using test benches written in SystemVerilog. These tests were designed to simulate real-world operating

conditions, ensuring that the fault injector module could reliably perform its intended functions within the PULP environment. To test the APB bus integration of the fault injector module, additional test benches were designed, ensuring that the fault injector module captured all the necessary configuration parameters for its ideal operations.

- Special attention was given to the driver logic that connects the fault injector module to target registers. This connection is vital for the precise injection of faults into the system. Through dedicated test benches, we validated the accuracy and reliability of these connections, ensuring that faults could be injected and retracted as designed, without unintended consequences.
- The validation of the new design elements and connections made using TCL scripting is incorporated into the TCL Script that makes this connection. A failure in the processing of the TCL script halts the script from executing along with its respective error message. This error message should be used to correct the script and the process can be restarted again.

6.2 Target registers

The registers that were targeted for evaluation of the fault injector module are from the execution stage and the Instruction Fetch (IF) stage of the cores of the PULP architecture. The parameter that dictates the number of target registers is changed before synthesis with the fault injector to create the number of output ports that will be connected to the output registers using TCL scripts.

6.3 Results

The results were evaluated in terms of FPGA resource utilization, timing, and energy efficiency. These key metrics provide us with information about how the fault injector module performs within the processor architecture. The design was synthesized at 1.02V, 25°C, typical corner, and targeting 125MHz operating frequency.

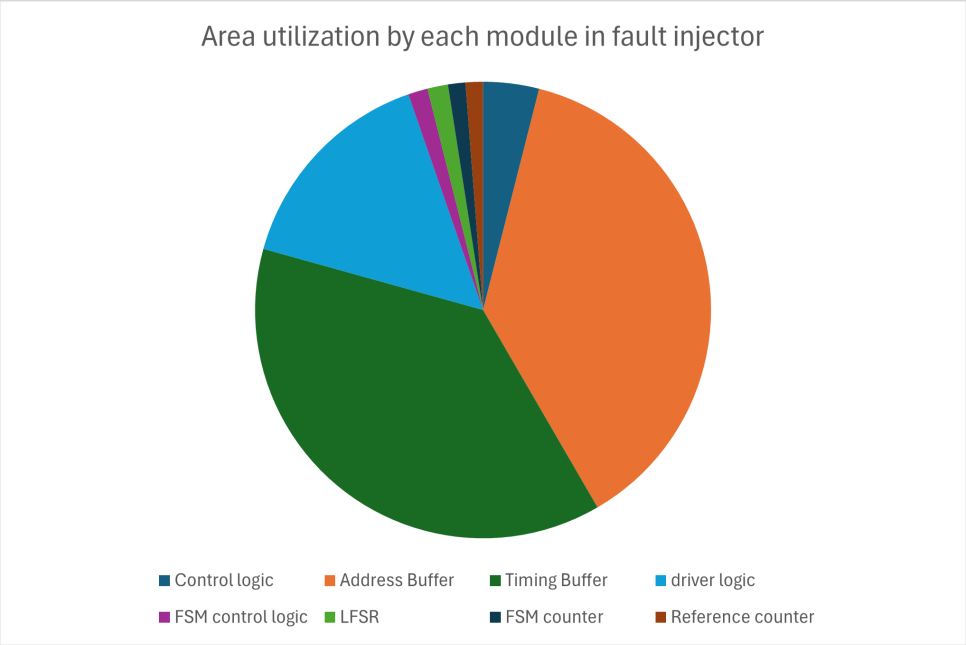

Figure 6.1 – Utilization of LUTs by the fault injector module

6.3.1 Timing and Utilization

The fault injector module was designed to work at the maximum frequency of 125MHz. In the PULP architecture, the fault injector module only interacts with the target registers, the fault injection modification done to the architecture is synchronized with the timing path of the target registers. Depending on the position of the target registers, the fault injector module does not limit the architecture's operation.

The evaluation of the utilization was done by connecting the output of the fault injector module to 200 registers of the prefetch buffer under the Instruction Fetch (IF) pipeline stage of the PULP architecture's core. In this work, the fault injector module's output ports are connected to the target registers through newly introduced design elements. As discussed in section 4.10, the newly introduced design elements require more CLB LUTs to be utilized. Fig. 6.2 shows the number of CLB LUTs utilized by the prefetch buffer in the IF stage of a PULP's RISC-V core before and after the addition of new design elements. The new design elements add an additional overhead that is equal to 1.6 times the unmodified prefetch buffer. The FPGA resource utilized by the fault injector module is 0.74% of the total available resources. Fig. 6.1 shows the area utilization of each sub-module of the fault injector module. It can seen that most of

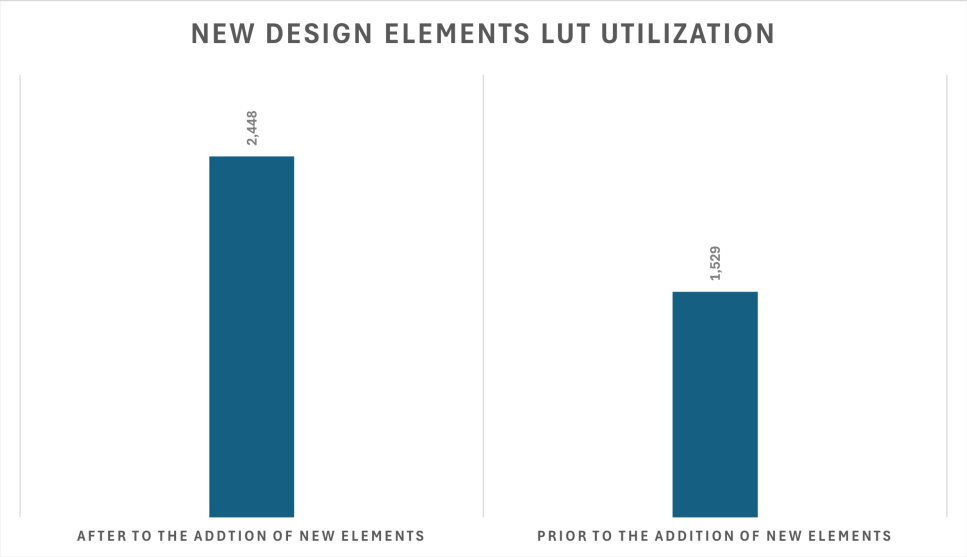

Figure 6.2 – Utilization comparison of LUTs

the resource utilization used by the fault injector module is its buffers. The buffer size can be adjusted during the synthesis process as needed. The rest of the sub-module utilization is only 21% of the total utilization of the fault injector module.

6.3.2 Power Consumption

Fig.6.3 shows the percentage of power, each sub-module of the fault injector module consumes. The power evaluation was done by performing a post-implementation simulation of the integrated PULP architecture. The total power consumed by the fault injector module is observed to be 1.407mW at 1.2V, 125MHz working frequency targeting 200 registers. As expected during the design phase, the power consumed by driver logic is the highest since it drives the fault into the targeted registers. This power also depends on the number of target registers, which is a parameter that can be modified during the synthesis phase of the design. The PULP architecture with an integrated fault injector, along with the new design elements consumes 0.394W of power. The power difference between bare PULP architecture and PULP architecture with integrated fault injector module is very insignificant.

6.3.3 Targeting Instruction Decode stage registers

The fault injector module was tested to target different registers from different parts of the architecture of the processor. The module is designed to have variable parame-

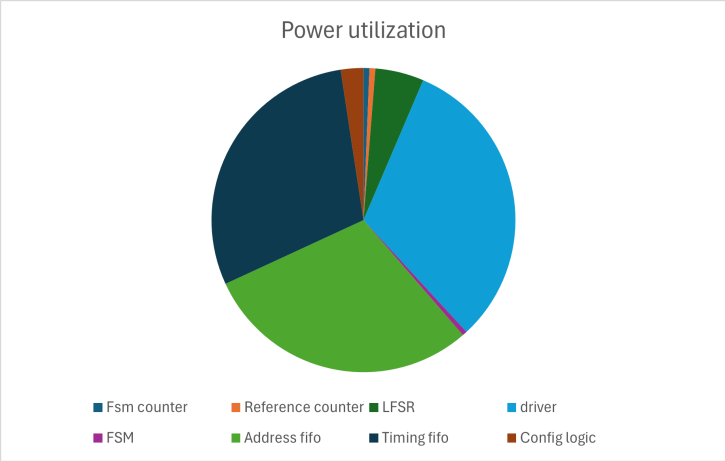

Figure 6.3 – Power utilization of the sub-modules of the fault injector

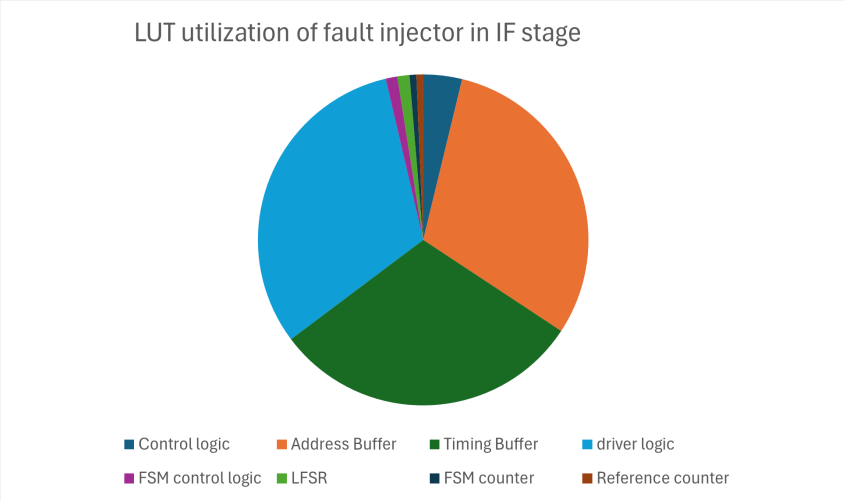

Figure 6.4 – LUT utilization of the sub-modules of the fault injector for 992 Target registers

ters that need to be changed during synthesis depending on the target registers. The number of registers targeted in this section is 992, belonging to the Instruction Decode (IF) Stage of the pipeline. The evaluation was done in a synthesized design at 1.02V, 25°C, typical corner, and an operation frequency of 125MHz. Fig. 6.4 shows the LUT utilization of the fault injector targeting 992 registers of the IF stage. As expected from the previous results, the buffers utilize the most LUTs. A key difference to be noted is the utilization done by the driver logic of the fault injector module. Since it has been scaled up to accommodate a large number of registers, the utilization of the driver logic seems to have increased. Fig. 6.5 shows the comparison between the utilization of LUTs for different target registers. It can be noted that only the utilization of the driver logic increases while the rest of the logic remains almost unchanged.

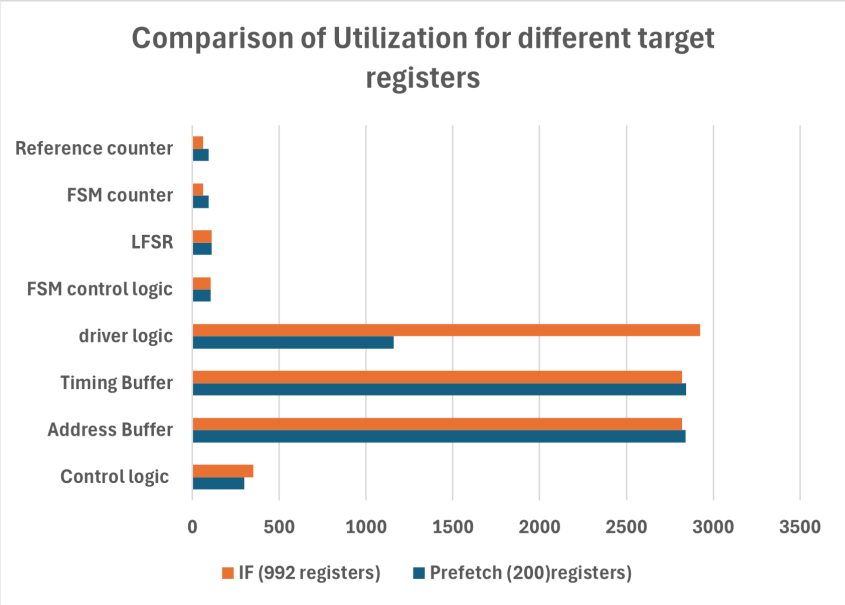

Figure 6.5 – LUT utilization comparison of the fault injector for 992 and 200 Target registers. The X-axis shows the number of LUTs utilized

The power utilization of the fault injector module increases as the number of target registers increases. The total power consumption of the fault injector module for targeting the IF stage registers is 1.71mW, which is 1.2 times higher than the power consumed by the fault injector targeting 200 registers.

From this analysis, it can be seen that the LUTs utilization and power consumption of the fault injector module depend solely on the number of target registers. The number of target registers can be modified, the fault injector module can be used to test small parts of the design if the overhead incurred by the fault injector module exceeds the limit of Device Under Test.

Chapter 7

Future improvements

The simulation of faults in architecture is the first step towards making it resilient to the faults that can occur in the harsh environment of space. The integration of the fault injector module was tested only on the PULP architecture. To further consolidate the versatility of the module designed in this work, it can be tested with other processor architecture families with scalable peripheral bus interfaces.

7.1 Data Collection on Fault Propagation

This work can be further expanded by gathering data on the registers of the PULP architecture. This part is done in this work, but not for all the registers. Creating a systematic approach to collect and analyze data on how injected faults propagate through the PULP architecture. This would involve monitoring and logging to trace the path of the faults from the faults injection points to observable outputs.

Utilize the data collected from fault simulations to identify which registers have a direct impact on the output. By understanding which registers are critical for maintaining minimal errors in the output, the fault injector module can be refined to target these registers preferentially. This targeted approach can significantly enhance the effectiveness of testing, allowing for more focused and efficient fault injection campaigns.

Based on this collected data, a predictive model can be developed that can estimate the fault tolerance of different components and configurations within the PULP ar-

chitecture. These models could guide the design of future iterations of the systems, focusing on enhancing resilience where it is most needed.

7.2 Adaptive changes to the architecture

The data from the fault injector module, allows us to identify mission-critical registers. With this processed data, The following strategies can be incorporated into the design and operation of satellite computing platforms.

- **Enhanced Error Detection and Correction (EDAC) Techniques:** Implement advanced EDAC techniques that can detect and correct errors that can occur in stored data. Depending on the criticality of the registers, we can use a variety of EDAC techniques like parity checks that can detect single-bit errors to more sophisticated systems like hamming codes, Reed-Solomon codes, and Convolutional codes that can detect and correct multiple-bit errors. This improves the resilience of mission-critical registers with little modifications done to the architecture[1].
- **Incorporation of Radiation Tolerant Components:** Where possible, radiation-tolerant components designed specifically for space applications can be used. While this approach may increase costs, it significantly enhances the reliability and longevity of satellite systems.[1]
- **Redundant System Design:** Resilience strategies like Triple Modular Redundancy (TMR) for critical register sets can improve the resilience of the architecture. TMR involves tripling the hardware and voting on the output to mitigate errors caused by radiation-induced faults. Furthermore, designing systems with redundant paths for critical functions can ensure correct operation if one path is compromised due to a fault, an alternative path can maintain system operations[7].

Incorporating these enhancements into the fault injector module will undoubtedly incur additional costs, both in terms of development and operational overhead. However, the cost must be weighed against the criticality of the application. For satellite computing platforms, ensuring resilience against space radiation is paramount, and the fault injector module plays a crucial role in this aspect.

While the adaptations may increase the overall cost, it is essential to consider the cost-effectiveness compared to space-based radiation-hardened processors. These specialized processors are inherently expensive and offer lower performance relative to their Commercial Off-The-Shelf (COTS) counterparts. The data gathered from this work can affect how the architecture can be modified, analyzing this data will provide crucial insight into the adaptions that need to be done to make the COTS processor resilient and ensure that it still outperforms the traditional radiation-hardened processors providing reliable operation at a fraction of the cost.

Chapter 8

Conclusion

In this study, we have presented a novel methodology for injecting faults into any processor architecture with a Peripheral Bus. The fault injector module was designed to inject faults into various parts of the processor architecture. Additionally, the fault injector module's versatility simplifies iterative processes such as testing and synthesis, streamlining the overall workflow. The fault injector module was integrated into the PULP architecture using the APB interface and targeted the Prefetch Buffers and Decoding registers of the IF and ID stages respectively. The PULP architecture was used as a test device, but the fault injector can be scaled to any processor architecture that supports the peripheral bus interface. It features very low LUT utilization and power consumption while still being able to target a large number of registers. The integration of a fault injector module into a processor architecture requires very little RTL code modification and uses TCL scripting to target registers in the design, enabling fast and reliable iterations for testing the different part of the architecture.

Bibliography

[1] Eric Cheng, Shahrzad Mirkhani, Lukasz G Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R Stan, Klas Lilja, Jacob A Abraham, Pradip Bose, et al. Clear: Cross-layer exploration for architecting resilience-combining hardware and software techniques to tolerate soft errors in processor cores. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[2] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. Open-source ip cores for space: A processor-level perspective on soft errors in the risc-v era. *Computer Science Review*, 39:100349, 2021.

[3] Ran Ginosar. Survey of processors for space. *Data Systems in Aerospace (DASIA). Eurospace*, pages 1–5, 2012.

[4] Sukrat Gupta, Neel Gala, GS Madhusudan, and V Kamakoti. Shakti-f: A fault-tolerant microprocessor architecture. In *2015 IEEE 24th Asian Test Symposium (ATS)*, pages 163–168. IEEE, 2015.

[5] Jiemin Li, Shancong Zhang, and Chong Bao. Duckcore: a fault-tolerant processor core architecture based on the risc-v isa. *Electronics*, Volume: 11(1):page: 122, year: 2021.

[6] Michael Rogenmoser, Yvan Tortorella, Davide Rossi, Francesco Conti, and Luca Benini. Hybrid modular redundancy: Exploring modular redundancy approaches in risc-v multi-core computing clusters for reliable processing in space. *arXiv preprint arXiv:2303.08706*, 2023.

[7] Satyam Shukla and Kailash Chandra Ray. A low-overhead reconfigurable risc-v quad-core processor architecture for fault-tolerant applications. *IEEE Access*, 10:44136–44146, 2022.