Politecnico di Torino

Politecnico di Torino

A.a. 2024/2025

Graduation Session September 2025

Accelerating the reliability assessment of hardware accelerators through emulation using hyperscale systems

Master of Science Thesis

Professors:

Matteo Sonza Reorda

Juan David Guerrero Balaguera

Candidate:

Berkay Demir

Abstract

Faults within the hardware accelerators compromise the reliability of the entire system. In the case of a system failure, the more complex the system becomes, the more difficult it becomes to diagnose the origin of the faults. The location of the fault also becomes a critical issue. A fault located near or on the critical path results in a much higher divergence from the correct result and optimal performance. To analyze hardware accelerators for specialized needs, they are tested using different fault models to measure the effect on the performance of the system during the manufacturing stage.

This thesis provides a comprehensive analysis on the emulation of hardware accelerators with fault injection capabilities on an FPGA fabric. In order to increase time efficiency, the control of the fault injection campaign is done through the design of a dedicated hardware controller. The integration for fault injection is developed for fault characterization within reasonable time frames, since software control of the module spends critical execution time.

The research methodology follows a systematic approach, beginning with analysis and background research on the saboteur insertion framework within the hardware accelerators to develop fault injection capabilities. The framework of saboteur circuits is meticulously implemented in an interconnected chain architecture. This saboteur structure includes multiple fault models such as stuck-at-1, stuck-at-0 and transient faults. The configuration is made through a shift register scan chain, pushing an array of values through them inside the saboteur circuits for mode selection and bit activation.

The second contribution is the development of a fault injection controller circuit that decreases the fault emulation time. The connections for the control of the circuit are designed according to the specifics of the Wishbone B4 bus interface. The Verilog controller is developed to manage the modification and configuration of the shift registers and the activation of entire scan chains on different hardware cores, utilizing the same fault configuration of the scan chain shift registers. Through its finite-state machine, the controller handles status reporting through its bus interface, fault injection through its serial data output, and immediate and delayed fault activation for the configuration of the entire scan chain. The controller utilizes memory-mapped registers in order to program and modify different chain lengths for different hardware, making its use much more viable for hardware with the same scan chain configuration, with less memory use.

The third contribution is the integration of a hardware accelerator with an inbuilt scan chain to the saboteur control circuit and emulation on the HyperFPGA platform. The HyperFPGA system features hardware resources that are necessary

for the emulation of the system. The connection between the system and the hardware is made possible through ComBlock communication interface, an IP core that connects the gate array's server programming to the stereo-core hardware. The ComBlock provides multiple communication interfaces that include register-based, a dual port RAM and asynchronous First In, First Outs (FIFOs) for streaming data transfers.

The last contribution is to emulate 4 other hardware accelerators for evaluation of the results to be used as a benchmark. The hardware accelerators to be used in the test are a Tensor Core Unit, a Stereo Core accelerator based on census transform, a CORDIC Special Function Unit core and a Special Function Unit that calculates trigonometric functions. A brief overview of the benchmarks and results of these is analyzed, while giving specifications on which parts of the chain the faults are implemented, the difference between them and effects of different fault models on the results are evaluated. The results show a significant boost in speed compared to traditional fault injection and highlights the efficiency of the controller.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ever increasing diversity within the modern computing systems took a different path from central processing units to more specialized hardware accelerators to meet the demanding performance and energy efficiency requirements of emerging applications. From autonomous vehicles processing stereo vision data in real-time to data centers accelerating machine learning workloads, hardware accelerators have become essential components of computing infrastructure. The complexity of these systems that stem from ever more advancing needs increased the need to test the reliability of these specialized circuits.

In order to understand how a system works and how reliable it would be, engineers have tried to model the defects that can arise within the systems. As more and more circuits are manufactured and used, the assessment of reliability became even more necessary. These defects would be named as faults within the hardware. Reliable operation in the presence of hardware faults has been tested in manufacturing processes for decades.

The basic way to fault models within a system would be to use software simulators running VHDL or Verilog hardware in order to evaluate the effects without going through emulation on programmable field arrays or gate arrays. This procedure provides basic observability of fault effects on the system reliability; however, the actual hardware would be more prone to fault defects as simulation works on perfect environments, not taking into account any component effects and real-life scenarios such as clock jitters or thermal effects. The second limitation on characterizing the effects of faults is the simulation speed. In a comprehensive reliability test, different fault models would be used to test out different fault injection places that can cover up to a thousand potential injection points, leading to multiple hours of simulation time. Simulation time that is not feasible in real-life scenarios where some high-level systems can reach millions.

1.1 Goal

This thesis establishes a practical, scalable methodology for reliability evaluation and simulation time of hardware accelerators, contributing practical hardware tools to advance the field of reliability evaluation. The developed framework enables faster transition from simulation evaluation to emulation on hyperscale FPGA systems. The goal is to accelerate the reliability assessment of hardware accelerators through newly developed hardware in emulation. In order to achieve this, a controller module has been developed. The dichotomy of slow software, fast hardware is a specific issue. Rather than the modification and activation being achieved through the software, this controller serves as a bridge in emulation timing reduction. This way, achievement of the speedup of the fault injection process increases dramatically.

1.2 Structure Of The Thesis

This thesis is organized in 6 chapters. The first chapter is the introduction, giving a fairly well established beginning to the thesis. This chapter also introduces the motivation, establishing the need for this work, and the general structure of this thesis.

The second chapter provides the necessary theoretical foundation needed for understanding the work done in this thesis.Starting with an explanation of faults and fault models, focusing on stuck-at-1, stuck-at-0, and bit-flip models, it further dives deep into their theoretical models to showcase how they are affecting a system in their respective models in general. Furthermore, this chapter explains the fault injection scan-chain design and the hardware for disrupting the output of basic nets. The theoretical model of the circuit is explained methodically for further understanding how fault methodology works in this thesis. Finally, this chapter dives into the hyperscale fabric we emulated our design in, the HyperFPGA design, a cloud-based FPGA infrastructure featuring Zynq UltraScale+ devices provided by Multidisciplinary Laboratory (MLAB) and the ComBlock communication infrastructure developed by MLAB/ICTP that provides register-based control for efficient software-to-FPGA communication.

The third chapter presents the design of the saboteur controller, the critical hardware module that is specialized in speeding up the fault injection and activation. The chapter provides the necessary explanations for the development of such a module, by explaining how the mechanism works with small segments of code attached to it. Utilizing the Wishbone B4 BUS interface, it implements a slave interface with a master connection coming from the software. Using continuous interaction from the software by synchronized fault injection using 32-bit registers,

data streaming to the fault interface increases in speed because of the fast hardware clock, rather than slow software interaction. The nine state machine is thoroughly analyzed, giving the user an in-depth understanding of the algorithm behind the single cycle precision fault injection and activation. Throughout this chapter, Verilog-based timing diagrams are provided to give a better understanding to the reader.

The fourth chapter provides the entire software and hardware configuration, detailing how the FPGA frame is used in order to emulate the system. The details of the system and the methodology behind the Python scripts are explained. The integration of the system on the HyperFPGA platform is given, detailing how ComBlock and the system clock are utilized to target the accelerators on a posit adder accelerator. The Jupyter network interface is explained on how to enable the fault campaign. The results of both the software simulation and hardware emulation are presented with tables.

The fifth chapter presents the comprehensive evaluation of the results using four different hardware accelerators as benchmarks to validate the framework's efficiency. Building upon the previous chapter, the benchmark accelerators are a Tensor Core Unit, a Stereo Core accelerator based on census transform, a Special Function Unit core that calculates trigonometric functions, and a Special Function Unit core that implements the general CORDIC algorithm. The results are presented in nine tables, diving from resource allocation, power usage to timing analysis.

The sixth chapter provides the conclusion and future work of this thesis. Starting with a small summary of the problems and achievements at hand, it concludes the previous work by summarizing the key results achieved throughout the previous chapters and explaining the limitations. Finally, ideas for future work are explained, such as software and hardware optimizations.

Chapter 2

Background

2.1 Introduction

In this chapter, background research for the development of the thesis will be provided, starting with a focus on the theoretical aspects of the faults within the circuits that will be instrumental in this work. To provide a comprehensive understanding of the work done in this thesis, a deep dive into the fault mechanism, the classifications of faults and the differences between them, and modeling methods will be thoroughly explained. Secondly, to demonstrate how modern design testing works, the scan chain architecture and the basic building blocks that are used will be explained. Lastly, to give a background on the emulation fabric, the hardware ComBlock for the communication between software and the hardware HyperFPGA for emulation will be explained in an explicit way.

2.1.1 Hardware Faults in Digital Systems

Fundamental Concepts and Classification

The ever increasing complexity within the systems, due to the shrinking transistor sizes, resulted in the need for much heavier testing to ensure reliable operation under variable situations. The concept of fault revolves around the idea of defects in the hardware operating cycle. A fault is a defect within the system hardware that causes unwanted responses and deviations from the expected result of a digital system. Faults are classified by their duration. Three categories that this work takes into account can be classified: permanent, intermittent, and transient faults. Permanent faults are, as indicated, lasting throughout the operational lifetime of the device and require physical replacement of the part or parts. They usually stem from hardware defects during manufacturing processes such as open and short circuits in the vias. Another common occurrence in which the fault occurs is

during electromigration, where conductor atoms break apart and migrate during high-current-density scenarios.

However, an intermittent fault is classified as a malfunction that occurs at irregular intervals in a device that can function regularly. Harder to diagnose because during the testing stages, the operation can commence and end fault-free. The most prominent reason for these types of faults is environmental effects such as the variation of temperature during operation. The transistor voltages within the system can fluctuate during thermal expansion, leading to operational failures. Because of this unpredictable nature, specialization and multiple runs for testings are usually needed for full fault coverage.

A transient fault is a malfunction that is relatively short within the system operation time. It can be seen in the circuits that utilize smaller architectures with fluctuating small voltage levels in sensitive nodes, mostly from electromagnetic interference. Mostly problematic for true data transmissions, these types of fault can disrupt the normal operation of digital hardware.

Fault Modeling and Fault Testing

During the testing phase of many hardware, in order to simulate the defects that will inevitably arise from faults, engineers have come up with a terminology in order to model the effect to be seen. Fault modeling is the development of intentional system failure in order to simulate actual physical defects within a system as a logical model. With the exponential growth of hardware in the 21st century, this resulted in development of different abstraction levels, mainly gate-level modeling and transistor-level modeling. The reason for the development of these levels stems from the tradeoff between high accuracy and computational difficulty. Transistor-level fault modeling has been the go-to for simulating faults within individual transistors. Stuck in a transistor model means a permanent conducting transistor or an open circuit transistor independent of the gate and drain voltage. The benefit of having a comprehensive fault analysis gives many statistics on effects of faults, however, this detailed analysis also comes at a computational expense. In order to find a medium, gate-level fault models are developed to provide a higher level of abstraction for analyzing faults on digital circuits. Rather than per transistor, the work details the logic gates and their changing responses. The most common among them is the stuck-at-fault model. During this work, this model will be extensively used to modify the circuit response to compare speed between simulation and emulation. For further analysis, the stuck-at concept will be extended to transition fault models in order to capture timing-related defects by modeling faults that prevent or delay signal transitions. Path-delay fault models consider the cumulative effect of small delays along critical paths that can cause timing violations without affecting the logical functionality.

Stuck-At Fault Model

The stuck-at-fault model is designed to fix a node to be permanently grounded (stuck-at-0) or supplied (stuck-at-1). It is the most common fault model to be used for it's simplicity to design and comprehensive and detailed simulation for behavioral results. In gate-level modeling, the fault is simulated to occur on the output of gates or in signal lines. In order to maintain simplicity within fault analysis, the assumption to be made is that only a single fault is active within the circuit. In a circuit with n signal lines, 2n stuck-at faults can exist. The difference between the fault-free model result and the injected fault result is then compared, giving a comparison metric that calculates the deviation from the result between all 2n faults within the system. In this way, the critical nodes that have a higher deviation rate can be observable. Thanks to modeling, in case of real-time fault, faulty gates, signal lines, transistors, and vias can be identified.

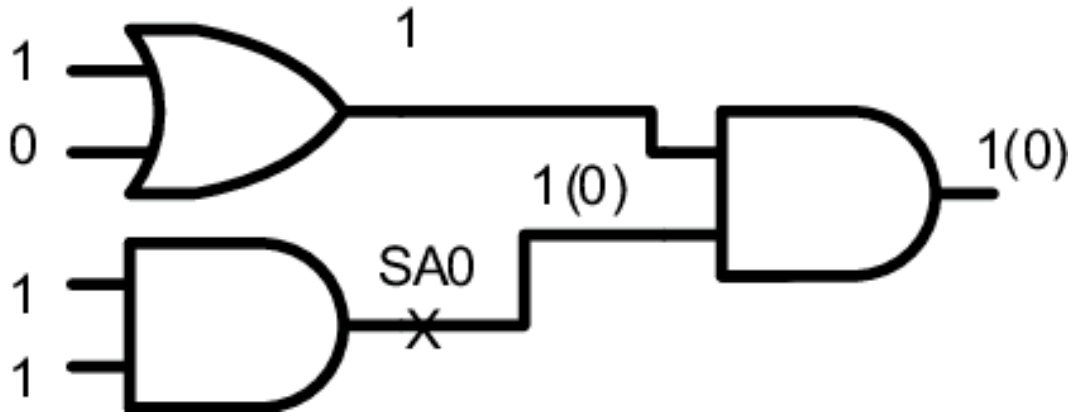


Figure 2.1: Stuck-at-0 model [1].

The current stuck-at-0 model shown in the Figure 2.1 takes the test sequence (1,0,1,1) to its ports. This results in a fault excitement on the signal. The AND gate 1 output expected to be 1, resulted in 0. The expected result on the AND gate 2 is 1, while the propagated result of the stuck-at-0 fault is observed on the output.

Bit-Flip Fault Model

The bit-flip fault model is a type of transient fault that inverts the bit value the signal is carrying through the gate. It is especially important during operations due to environmental factors and electromagnetic interference. There are 2 Bit-flip models, **Single Event Transient (SET)**, where the fault affects the combinatorial logic, and **Single Event Upset (SEU)** fault model that affects the flip-flops,

latches, and memory structures. Since it affects the memory structures, the result is persistent until overwritten. During operation, the bit values are complemented for a specific time, resulting in variable deviations from the intended output. Since **Single Event Transient (SET)** faults are temporary, unlike the stuck-at model, they can be masked during normal hardware operation before propagating to the observed output, making them much harder to evaluate. The shorter the duration of the flipped bit, the harder it is to see the disruption on the result. The clock cycle duration is imperative in this, since a longer clock cycle and shorter fault duration can result in deviations not being captured on the output of the flip-flop. For instance, Figure 2.2 shows the mechanism for a Single Event Upset (SEU) bit-flip simulation.

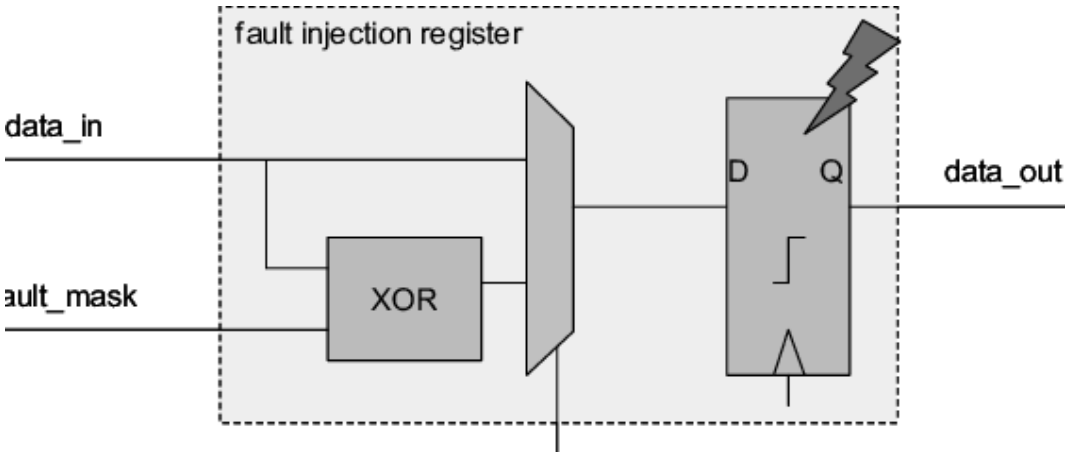


Figure 2.2: The simulation mechanism used for observing the effects of an SEU bit-flip [2]

This model is the third configurable fault model that is utilized within this work, next to stuck-at-0 and stuck-at-1.

2.2 Fault Emulation Using Saboteur Infrastructures

2.2.1 Basic Saboteur and Scan Chain

In this work, the hardware configuration is different from Design-for-Testability. Compared to the previous design utilized in manufacturing, this work focuses on reliability assessment. The architecture revolves around injecting faults for emulation, through addition of saboteur circuits into the design. In order to simulate this design, a basic saboteur circuit is utilized. The architecture of this

circuit revolves around a simple multiplexer-based architecture that is utilized to alter the output of nodes.

State-of-the-Art in Saboteur-Based Fault Injection

Multiple approaches have been made to the saboteur-based fault injection, with various architectures for the design having been proposed. The architecture has been utilized in the work of Jenn et al. [3], when the modification of the HDL level has been made possible by the MEFISTO tool. The architecture of this tool enabled HDL-level manipulation for saboteur injection for simulation. The architectural design came with the overhead of simulation time increase. FPGA-based fault emulation has been utilized in the work of Civera et al. [4]. This gave progress to fault injection from simulation only to hardware emulation using FPGA on microprocessors. The fault injection achieved a significant speedup over simulation while maintaining control over fault location. FPGA-based fault emulation was further developed during the work of Alderighi et al. [5]. The demonstration of Single Event Upset (SEU) emulation in SRAM-based FPGAs was made possible with the development of the FLIPPER platform. The results of this work highlighted the trade-off between fault coverage and area cost, proving that the strategic placement of saboteurs is of the utmost importance. More recent work has focused on optimizing fault injection campaigns. The saboteur-based scan chain infrastructure implemented in this thesis is based on the work of Sensoz [6], who developed a comprehensive FPGA-based fault injection framework for the evaluation of the reliability of stereo-vision based accelerators.

The architecture includes a multiplexer-based saboteur unit that selects the fault model to be injected based on the control bits, a shift register that saves the saboteur's mode, and the enable logic that determines whether the faults are to be activated or to be used as a normal output.

The control logic uses the two most significant bits of the scan chain to alter the mode in progress. All the fault models, stuck-at-0, stuck-at-1, bit-flip can be utilized. In this way, hardware gate-level architecture is utilized.

The scan chain integrates shift registers with saboteur circuits to create a configurable fault injection network. Each saboteur module in the digital hardware design includes an associated shift register that stores its configuration. The serial shift register utilized in the scan chain accepts the serial data input and pushes it forward through parallel outputs defined by their width. These shift registers are connected in a daisy-chain fashion. In this way, the hardware forms a continuous scan chain through the circuit, connecting one to another while allowing for the activation of different saboteur circuits. The operation to activate the scan chain starts from the enable signal being activated. During this, the controller shifts exactly the length of the configuration data through the scan-chain, then disables

the enable when configuration is done. The saboteur operates as a two-stage multiplexer circuit. The first stage (controlled by `i_ctrl[1:0]`) selects the fault mode ,and the second stage (controlled by `i_en`) determines whether the fault is active or bypassed. This scan chain architecture works in two phases. The first phase is the configuration, where the fault patterns for the activation of the faults and the fault mode to be configured are shifted through the scan chain (controlled by `i_en`). The second phase is the activation phase, where the configured faults are activated by `i_TFen`).

2.2.2 FPGA-Based Emulation Platform and Communication Infrastructure

In this chapter, the explanation of FPGA architecture and emulation of ComBlock. FPGA platform provides us with the necessary capability to emulate the designed hardware with a ComBlock providing the efficient communication between software and hardware execution. The integration between them ensures the transition from simple software simulation to hardware-accelerated fault injection, to test the reliability in real-life trials that are necessary for developing comprehensive reliability evaluation.

HyperFPGA Platform

The HyperFPGA platform is a cloud-based FPGA emulation fabric accessible via remote server [7].This FPGA platform provides hardware support for testing newly implemented modules with multiple resources for extended projects. The platform uses Trenz Electronic System-on-Modules featuring the AMD/Xilinx Zynq UltraScale+ MPSoC, which integrates a quad-core ARM Cortex-A53 processor with substantial FPGA resources: over 192,000 logic cells, 728 DSP slices, and 16.9 Mb of Block RAM [7].

The platform can be accessed and modified through the Jupyter Notebook that provides accessible bitstream manipulation. The entire hardware infrastructure includes these features, as described in the document of [8]:

- Xilinx ZYNQ UltraScale+ MPSoC, U1
- 2-Input AND Gate, U39
- Red LED (DONE), D1
- 256Mx16 DDR4-2400 SDRAM, U12
- 256Mx16 DDR4-2400 SDRAM, U9
- 256Mx16 DDR4-2400 SDRAM, U2

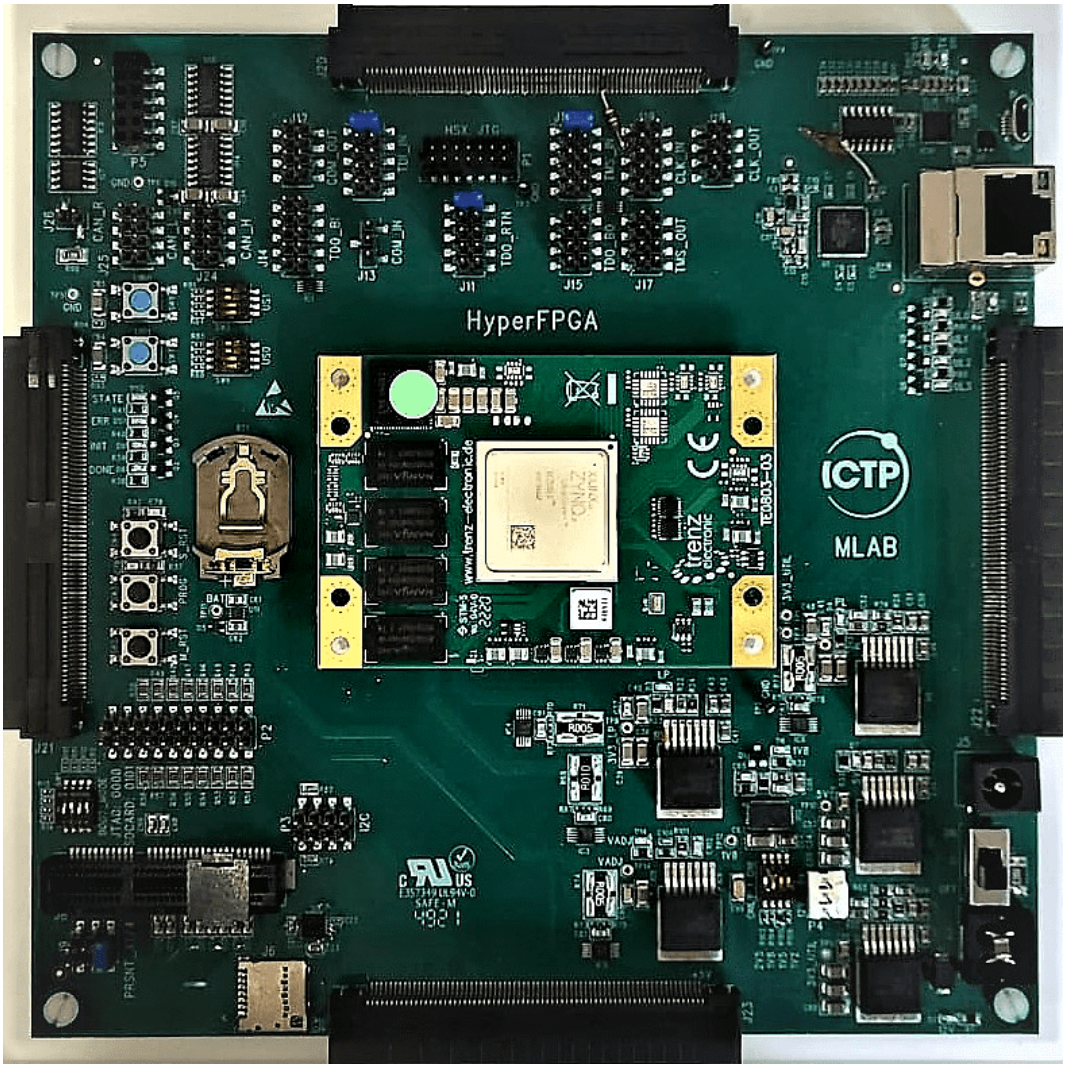


Figure 2.3: The HyperFPGA [7].

- 256Mx16 DDR4-2400 SDRAM, U3
- 12A PowerSoC DC-DC converter, U4
- 1.5A LDO DC-DC converter, U10
- 1.5A LDO DC-DC converter, U8
- Voltage monitor circuit, U41
- 0.35A LDO DC-DC converter, U26

- 0.35A LDO DC-DC converter, U27
- Ultra fine 0.50 mm pitch, Razor Beam™ LP Slim Terminal Strip with 160 contacts, J3
- Ultra fine 0.50 mm pitch, Razor Beam™ LP Slim Terminal Strip with 160 contacts, J1
- Ultra fine 0.50 mm pitch, Razor Beam™ LP Slim Terminal Strip with 160 contacts, J4
- Ultra fine 0.50 mm pitch, Razor Beam™ LP Slim Terminal Strip with 160 contacts, J2
- 4-channel programmable PLL clock generator, U5
- Low-power programmable oscillator @ 25.000000 MHz, U5
- Low-power programmable oscillator @ 33.333333 MHz (`PS_CLK`), U32
- 256 Mbit serial NOR Flash memory, U7
- 256 Mbit serial NOR Flash memory, U17

ComBlock Communication Infrastructure

The ComBlock infrastructure is the basis for communication between the software and the hardware connection. ComBlock is an open-source IP core that is jointly developed by the MLAB of Italy and the INTI of Argentina, released under the BSD 3-clause license [9]. It is specifically implemented to simplify the operation of MLAB projects specialized in FPGA to/from PC data transmissions.

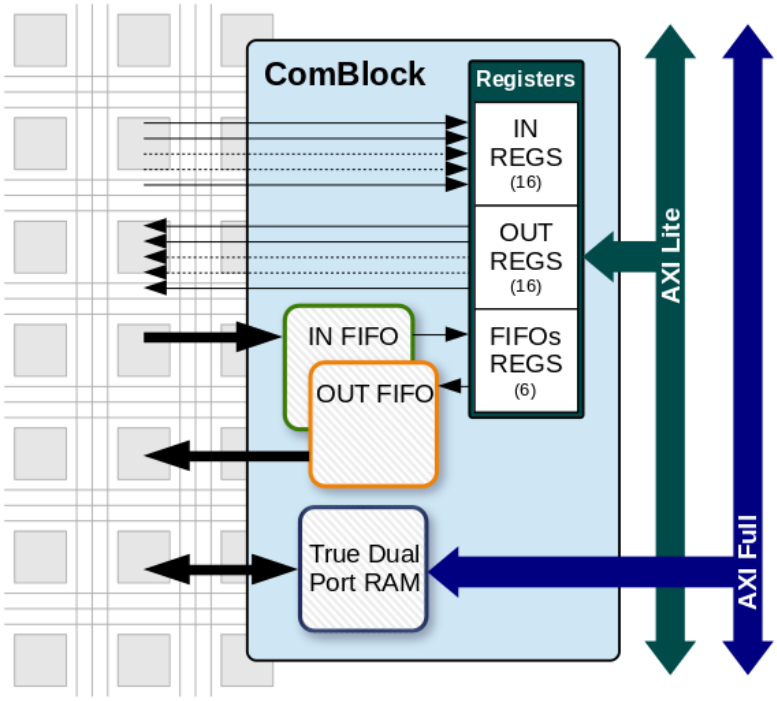


Figure 2.4: Comblock model [9].

As shown in Figure 2.4, ComBlock provides configuration for three interface types [9]:

- **Input-output registers:** Up to 16 registers with configurable widths of 1 to 32 bits for control signals and status reporting.

- **Dual-port RAM:** Memory interface with configurable parameters such as memory depth, address space, and data width.

- **Asynchronous FIFOs:** bidirectional FIFOs with empty, full, underflow, and overflow flags. Used for data transfer between the processor and the programmable logic.

The Vivado implementation employs the AXI Full protocol for RAM and the AXI Lite protocol for FIFO and register access [9].

The complete implementation within the Vivado is shown in Figure 2.5.

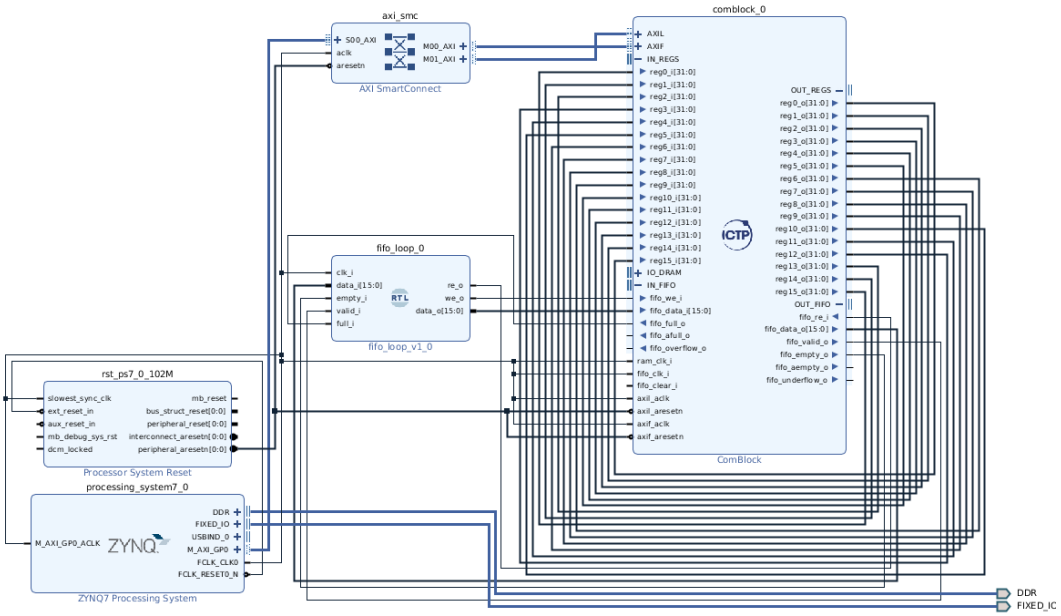


Figure 2.5: Comblock model [9].

2.2.3 Integration with Hardware Accelerators

In Vivado, the current implementation of Comblock and HyperFPGA demonstrates an adder and multiplier circuit connection using registers for input and output for testing purposes.

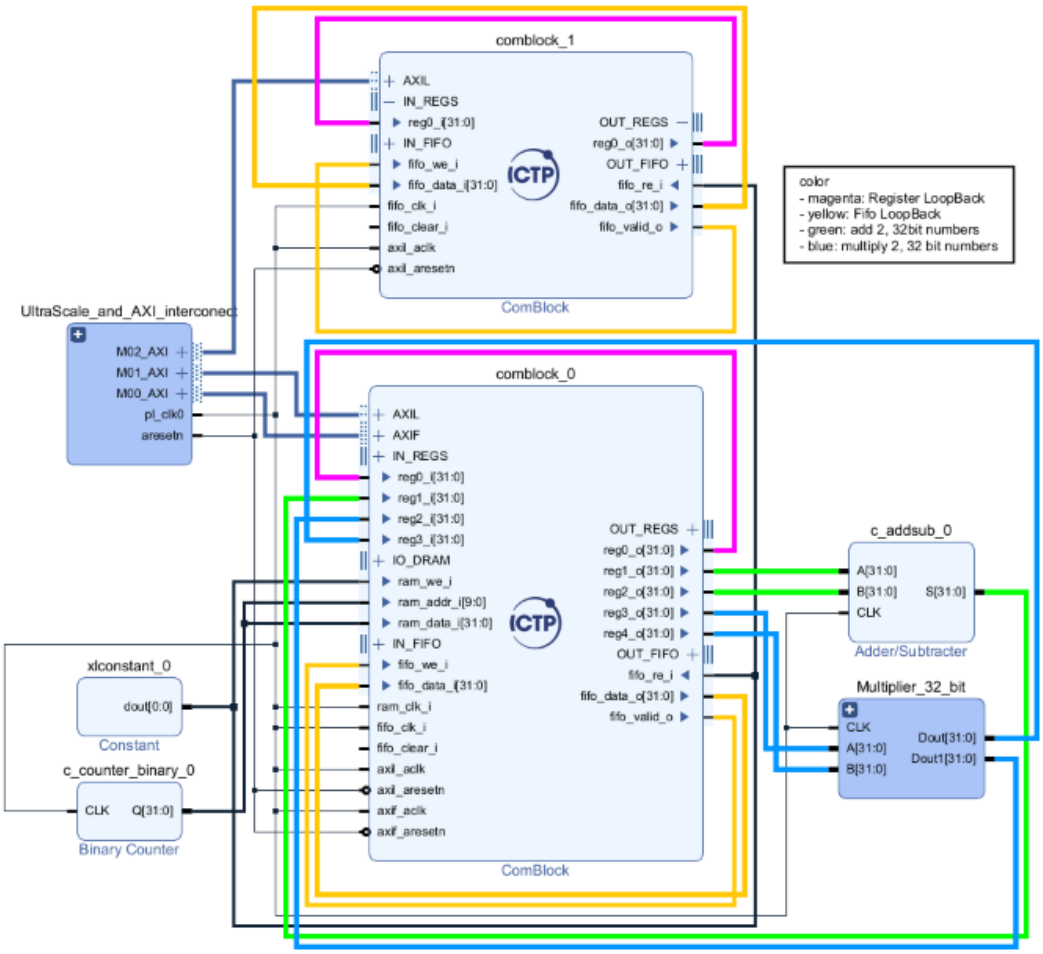


Figure 2.6: ComBlock baseline implementation.

This implementation in Figure 2.6 creates baseline hardware to modify to suit the needs of the hardware accelerator fault injection campaign. The ComBlock instances can be utilized to integrate any hardware accelerator needed using the register and FIFO input/outputs. The general idea is to use only a single instance of ComBlock and its input registers or the input FIFOs of the instance to enable the data transfer from the output of the hardware accelerator to be sent to the ARM Cortex-A53 processor for utilization. The data to be used can be hardware result operands or status flags read from polling, and integrated logic analyzer outputs for debugging. The register outputs or the FIFO output can be utilized to deliver operands, configuration parameters, or control signals from the ARM Cortex-A53 processor to the hardware accelerator.

Chapter 3

Design and Implementation of the Saboteur Controller

3.1 Introduction

The transition from simulation to emulation can be made in two primary ways. One is to directly control the injector through open connections of the scan chain, which the programming aspect of the entire fault injection would require the software programming in order to complete. The second is the development of a hardware controller that is programmed to ensure a quick transmission in clock speeds.

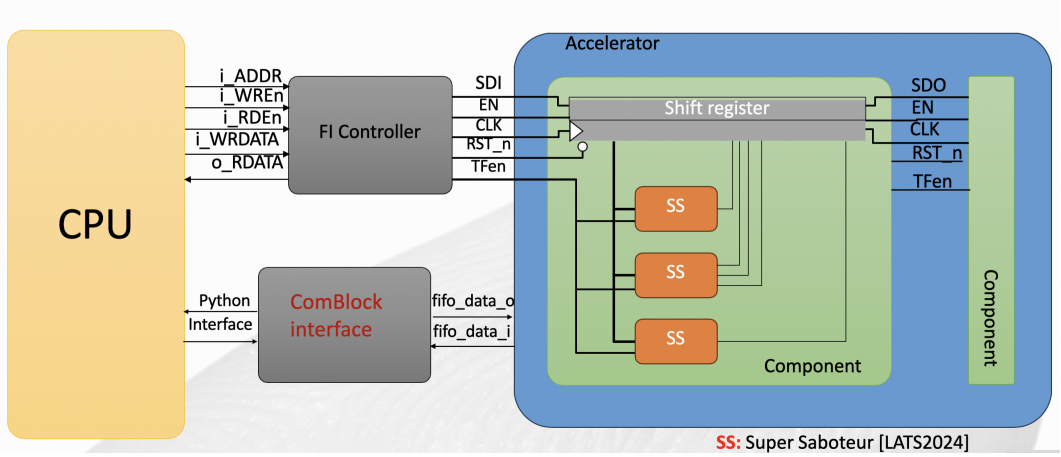


Figure 3.1: Proposed design for FI controller.

In the previous chapter, the theoretical foundation of faults and fault modeling, the architectural framework of HyperFPGA, and scan-chain design are explained. In

this chapter, an extensive analysis of the saboteur controller, a dedicated hardware module that serves as the critical interface between software-orchestrated fault injection campaigns and hardware-embedded saboteur infrastructure for emulation on FPGA is provided. The breakdown of this main component begins with its interface connection.

The reason for the development of the fault controller was the need for generalized hardware for continuous and serial application of faults without much modification to the software. Easily interchangeable between different accelerators, this module addresses the problems of changeable scan-chain sizes on different vision-oriented hardware with potential injection points on the scan chain up to thousands, accurate timing control with established Wishbone B4 BUS connections, and minimal interference between target hardware connections during synthesis and implementation. Through a minimalist but comprehensive finite-state machine, the controller achieves fault sequence injection and activation much faster, in microseconds, compared to emulating the scan chain through Comblock alone. This chapter explains the analysis of the module, beginning with its internal architecture, operation, and simulation timing analysis.

3.2 The Wishbone BUS Protocol and Finite State Machine

3.2.1 Fundamental Architecture and Design Philosophy

The Wishbone protocol is designed for flexible use of a communication interface within System-on-Chip interconnections [10]. The protocol uses a master-slave architecture. Data transmission relies on a handshaking mechanism between the master and the slave. The master asserts the cycle `CYC` and strobe `STB` signals of its ports, indicating the request for data transmission. The slave responds to this transmission request by asserting the acknowledge `ACK` signal. In this way, the reliability and precision of the system increase while the development time is reduced [10].

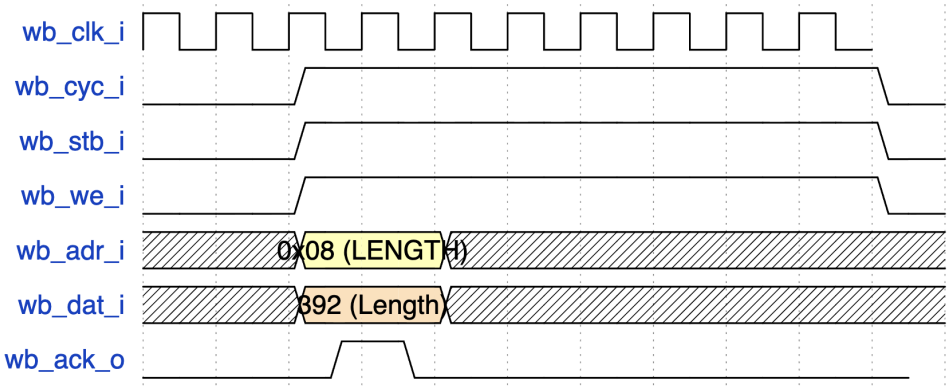


Figure 3.2: Basic write operation.

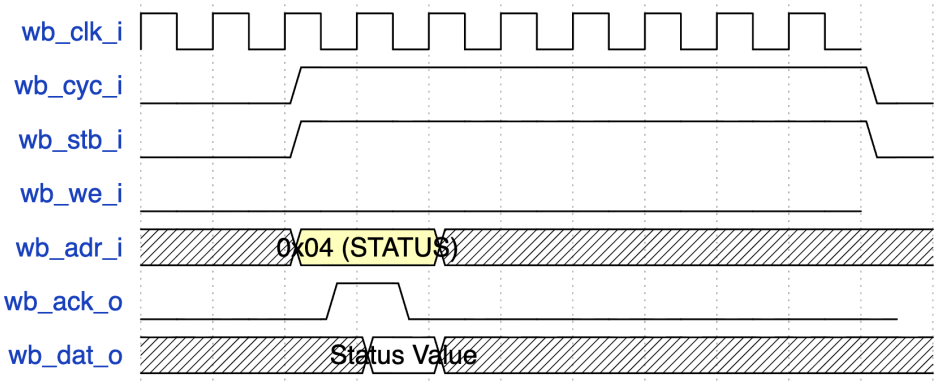


Figure 3.3: Basic read operation.

3.2.2 Controller Architecture and Interface Specification

Module Interface Definition

The controller module introduces two specific interfaces. First, the Wishbone B4 slave interface for communication between the master Comblock and the second is the fault injection control for saboteurs inside the scan chain. Thanks to both the BUS interface and the fault interface, it achieves consistent functionality with different types of hardware with variable chain lengths.

While the Wishbone interface strictly follows the rules for B4 implementation for robust communication by waiting for valid cycle and valid strobe inputs from the master and then sending acknowledge output back to the master, the fault

injection control interface generates all signals necessary for saboteur configuration and activation. The direct connection of `FI_CLK` to the system clock `CLK_I` eliminates clock domain crossing complexities while ensuring synchronous operation throughout the fault injection infrastructure. The `FI_TFen` provides a single-cycle pulse generation for accurate fault emulation, a critical requirement for modeling faults. The 32-bit data width architecture achieves alignment with standard embedded system architecture while the 8-bit address space allows memory space up to 256 register locations, far more than the controller's requirement.

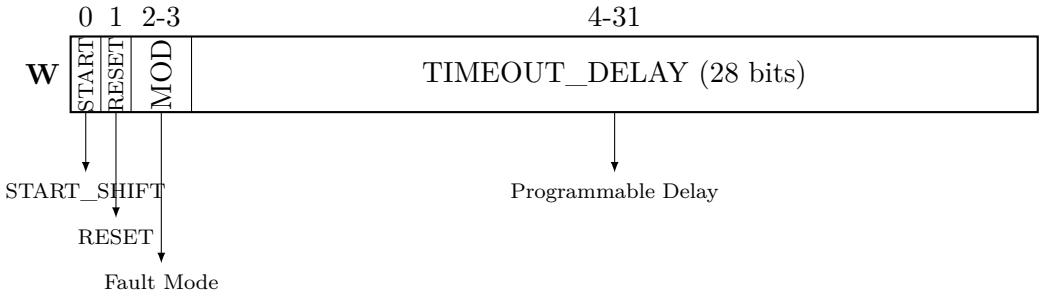
Internal Register Map

The controller uses four memory-mapped registers for providing total control over fault injection.

```
1 localparam REG_CONTROL_ADDR = 8'h00; // Control Register
2 localparam REG_STATUS_ADDR  = 8'h04; // Status Register
3 localparam REG_LENGTH_ADDR  = 8'h08; // Shift Length Register
4 localparam REG_DATA_IN_ADDR = 8'h0C; // Data Input Register
```

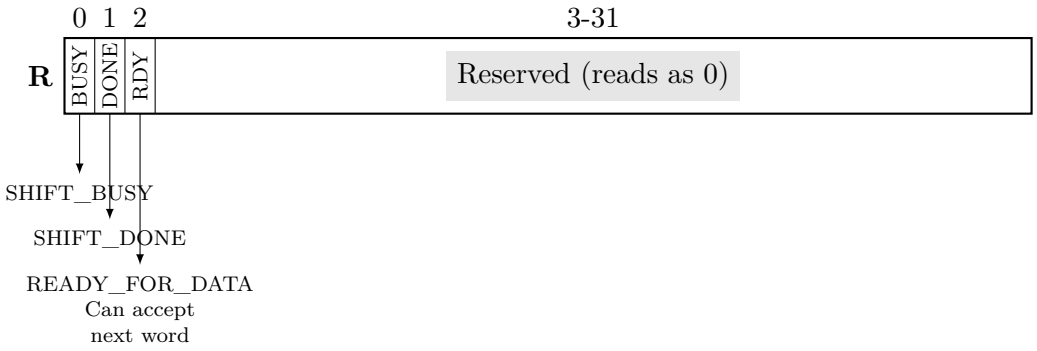
Listing 3.1: Controller Register Address Map

Figure 3.4: Bitfield Layout of the CONTROL Register (Address 0x00)



The control register at address 0x00 serves as the primary command register. The register is divided into 4 parts, with positions at: Bit 0 is used for `START_SHIFT`, used for initiating the configuration sequence and sets the internal flag of `start_shiftreq` to 1 that FSM clears during the configuration sequence.Bit 1 is used for resetting the scan chain. Provides the reset pulse for triggering the reset sequence. Bits 2 to 3 are used for defining the fault mode. Triggering the fault timeout request depending on the value to be written. Bits 4 to 31 are used for programmable timing delay for fault injection. Specifically designed for higher clock cycle values for a much later fault injection, specifically on transient faults for providing the single clock cycle faults. Ranges from 0 to 268,435,455 clock cycles.

Figure 3.5: Bitfield Layout of the STATUS Register (Address 0x04)



The status register at address 0x04 provides feedback about the controller state and operations in progress. Starting from bit 0, it includes the `STAT_SHIFT_BUSY` bit, where it shows the shift operation in progress or the controller being in idle state waiting for status reads. At bit 1, the `STAT_SHIFT_DONE` bit indicates whether the configuration of the scan chain is complete. It is a sticky bit, remaining set until cleared by a new `START_SHIFT` bit. At bit 2, `STAT_READY_FOR_DATA` bit is located. If set to 1, the controller is ready to accept the data word, and if not, it shows that it is currently shifting the loaded word.

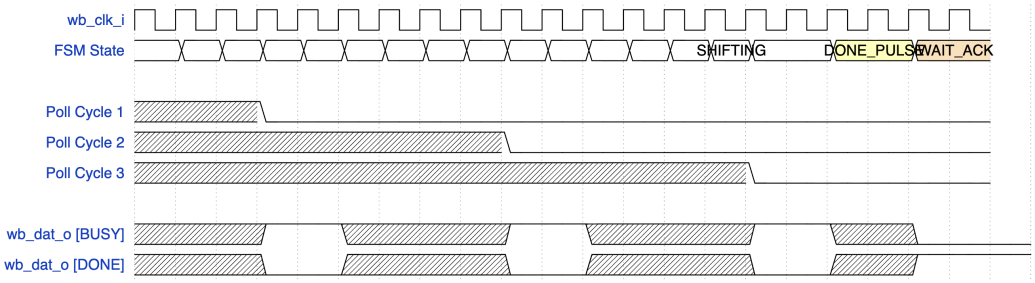
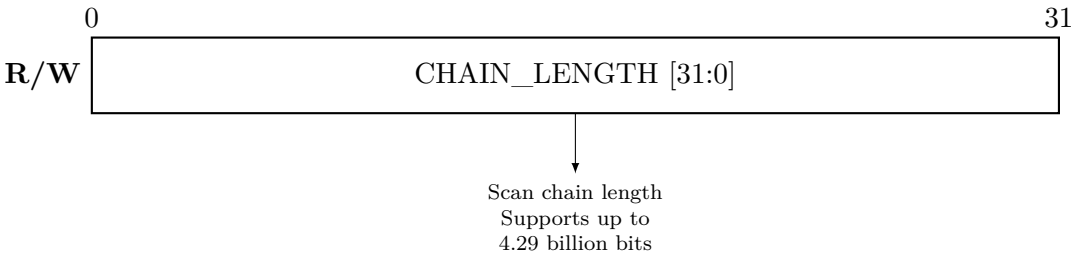


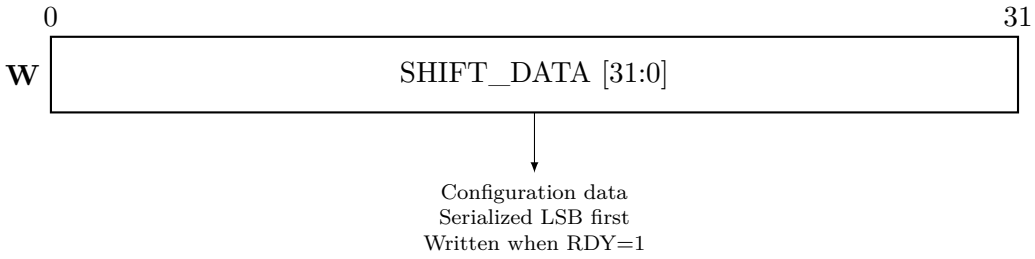
Figure 3.6: FSM states for status polls after shifting.

Figure 3.7: Bitfield Layout of the LENGTH Register (Address 0x08)



The length register named `CHAIN_LENGTH` at address 0x08 is a 32-bit register that stores the chain length of the shift register. This register provides support for chain lengths up to 4.29 billion bits. For simplification, it is separated from the status register for software interfaces and updates while requiring an additional write sequence as overhead.

Figure 3.8: Bitfield Layout of the DATA Register (Address 0x0C)



Data in Register named `SHIFT_DATA` at address 0x0C provides the streaming interface for configuration data. The register accepts a 32-bit data word that the controller serializes for transmission to the saboteur chain starting from the least significant bit. It is written after `status_ready_for_data` flag is set to 1.

3.2.3 Finite State Machine Design

The controller's FSM jumps through nine states that manage all the aspects from communication from master to the fault injection. The transactions between the states are configured through external commands such as Wishbone transactions' command bits and internal commands such as flags and expiring counters. As is common with finely developed state machines, sequential and combinatorial blocks are separated. State transitions occur at clock edges and resets on asynchronous `RST_I` and manage flags and registers, while the next state logic is combinatorial and updates the outputs based on the inputs. This helps avoid unnecessary latches within the logic.

The FSM states and their primary functions are as follows:

`FSM_IDLE`, the beginning state is the main state that is monitoring for commands, monitoring the flags and proceeding to the next state depending on the parameters. `FSM_WB_ACK`, the first state that acknowledges the Wishbone transactions. `FSM_LOAD_SHIFT` is the second state, responsible for initializing the shift operation. `FSM_WAIT_DATA` third state, main purpose is to wait for configuration data from software. `FSM_SHIFTING` is the fourth state. Responsible for the actual performance of active bit shifting. `FSM_SHIFT_DONE_PULSE` is the fifth state. It signals the configuration completion. `FSM_RESET_PULSE` is the sixth state. Responsible for generating

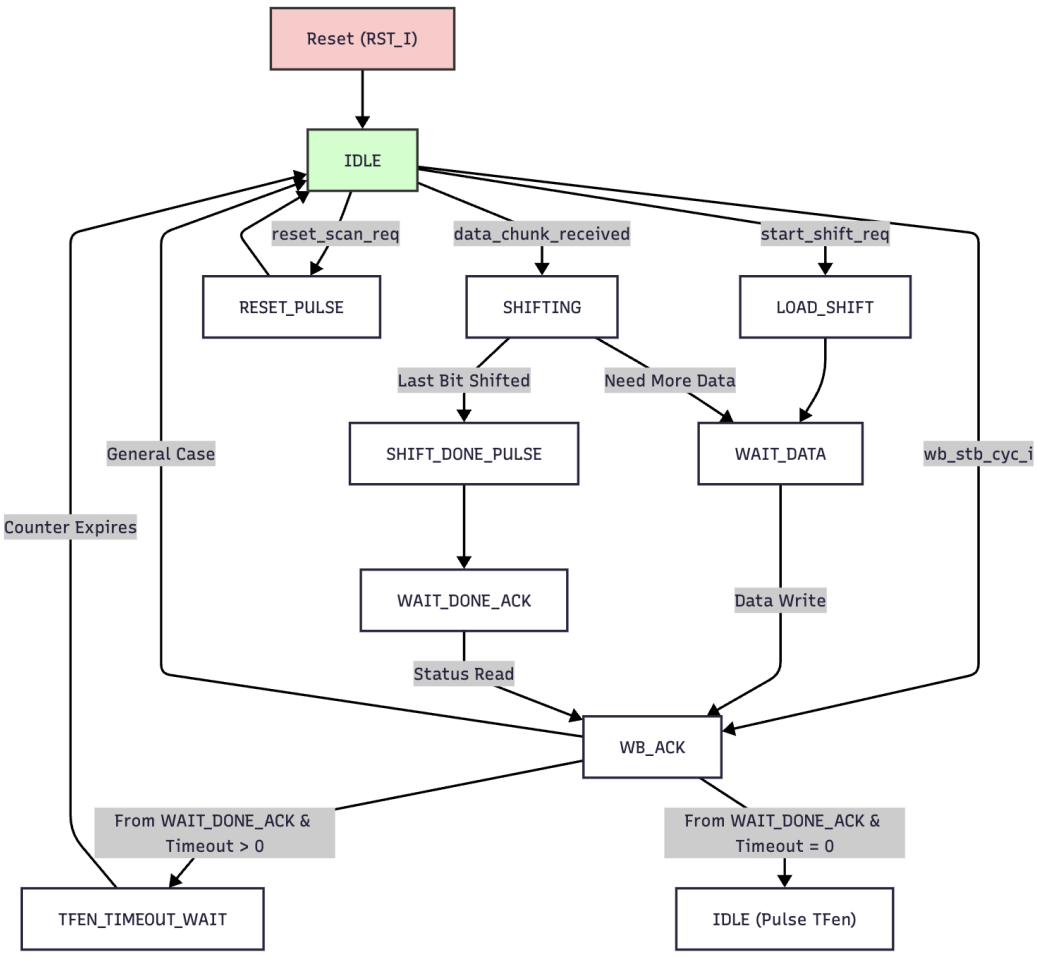


Figure 3.9: FSM design for saboteur controller

the scan chain reset for preparation of next testing. `FSM_TFEN_TIMEOUT_WAIT` is the seventh state. Responsible for the activation for fault enable signal depending on the timeout value configuration. `FSM_WAIT_DONE_ACK` is the eighth and final state. It ensures the final software acknowledgment.

3.2.4 Operational Modes and Timing Characteristics

Complete Fault Injection Flow

The fault injection process starts in software, determining the fault to be injected, how long the scan chain is, when and for how long it will be active in the hardware

accelerator. The information written from the software side will be sent to the saboteur controller through a sequence of handshake operations that will configure the controller. This controller will now serve as the bridge between the two fabrics.

Configuration Sequence

The configuration sequence, which includes the coordinated data stream coming from the master and the control of the shift registers in the scan chain, is the most complex part of this controller. The process initiates when the software writes the `START_SHIFT` bit in the control register. This triggers a cascade of hardware operations beginning with the FSM transitioning from IDLE to `LOAD_SHIFT` state. In this state, the `shift_counter` loads with the total chain length value from the length register, establishing how many bits must be shifted. The `bit_in_word_counter` resets to zero, preparing to index into the first 32-bit word. The `status_ready_for_data` flag asserts, signaling to the software that the controller is prepared to accept configuration data. These initializations are completed in a single clock cycle, after which the FSM transitions to `WAIT_DATA` state.

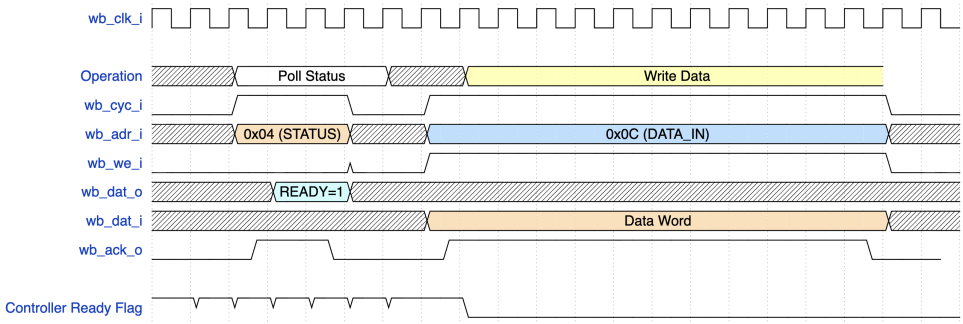


Figure 3.10: Configuration handshaking for data loading.

The handshake protocol prevents data loss during state changes from a fast clock to a slow software timing for the next data to be sent. The controller waits for the next data in `WAIT_DATA` state, while the software prepares the next data word to be sent. The dual operation then takes place, waiting for data while maintaining status requests. The state monitors the data register for data input while the controller responds to status register reads, allowing the software to notice the ready flag.

After the next data word is written to the data register, it is captured in the `current_shift_word` register, a temporary register written to during the Wishbone acknowledge. The internal flag is set to trigger the FSM to transition to the SHIFTING state.

The actual shifting occurs in the SHIFTING state. In order to send bit by bit, an index counter named `bit_in_word_counter` is sent to the `FI_SI` output. `FI_EN` signal is then asserted for shifting the scan chain by one position and capturing the `FI_SI` value. After this, the `shift_counter` decrements, the `bit_in_word` counter is incremented and moved to the next bit.

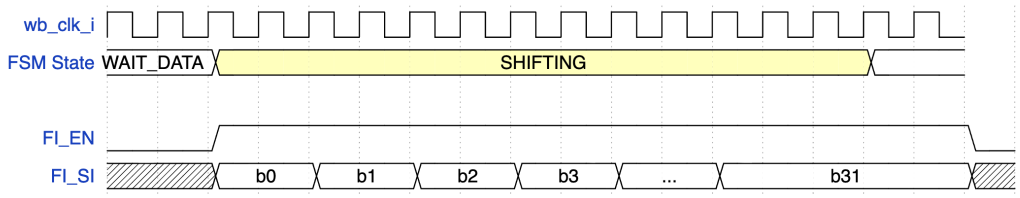


Figure 3.11: Activate data shifting.

After shifting the last bit of a 32-bit word, the controller decides whether to finish the shifting or return to `WAIT_DATA` for a next round of words by checking the `shift_counter`. Depending on the remainder of bits, if less than 32, the FSM returns. The specificity of "less than" is required because the chain length is not always a multiple of 32 bits. Depending on the remainder, the shift must stop after the total has been reached. Going through an example such as a 393-bit chain length. To shift, 13 words are required, with the last word being only 9 bits long in shift. The controller recognizes this by checking the remaining `shift_counter` and ignores the remainder of the bits from 9 to 31. The comparison logic can be found here:

```
1 if (shift_counter <= 32 && shift_counter > 0) begin
2     // Final word, potentially partial
3     if (bit_in_word_counter == shift_counter - 1) begin
4         // Last bit of entire chain
5         next_state = FSM_SHIFT_DONE_PULSE;
6     end
7 end
```

3.2.5 Completion Signaling and Acknowledgment

After completion of the shift, the remaining operations are timely activation of the faults and reset. This phase starts at the transition from SHIFTING to `SHIFT_DONE_PULSE` state. The reason for this state to exist is to configure a sticky bit to 1 until the next configuration and clear the `ready_for_data` flag, preventing further software writes through the BUS that would normally be ignored. In this

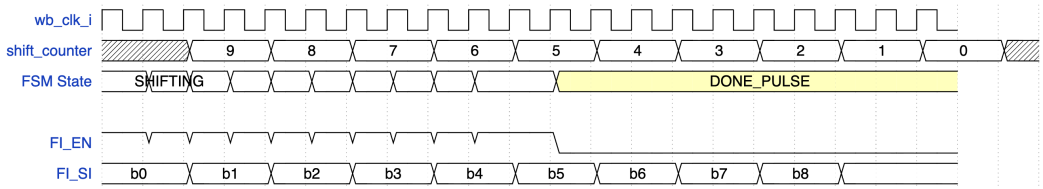


Figure 3.12: FSM state transition during partial shift.

way, software time is further reduced, which makes a much more optimal design without any overflow. Finally, the timeout value for the counter is written in the registers, preparing for the final fault activation phase after acknowledging that the fault injection is complete.

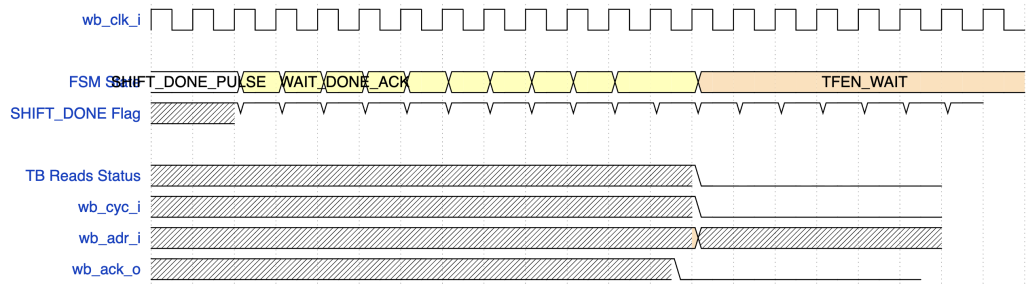


Figure 3.13: Synchronization for fault enable signal.

The state machine then enters the `WAIT_DONE_ACK` state, where the controller waits to read the acknowledgment signal from the software side, before continuing with the fault activation sequence. Wishbone BUS is used for any reads in the status register during which the controller waits for the software side observing the `SHIFT_DONE` signal being set, preventing any race conditions in the interface where the hardware continues with the state machine while the software is still processing the effects of the previous state.

3.2.6 Fault Activation Timing Control

The fault activation timing control transforms the state from waiting for acknowledgment to an actual timeout control. Providing flexibility for both transient faults and stuck-at faults with the timeout register, it keeps single clock cycle precision when actual fault injection occurs.

Although being programmable, the model of permanent faults of stuck at 1 and stuck at 0 is naturally taken into consideration from the beginning of implementation.

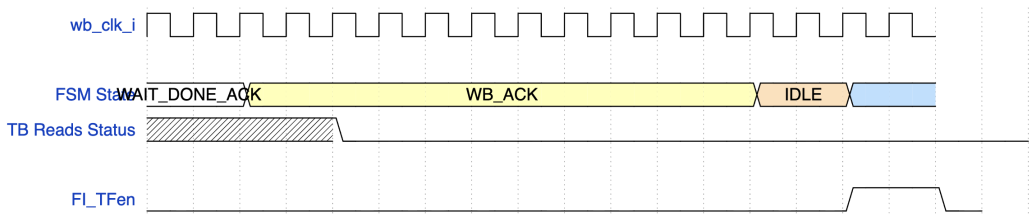


Figure 3.14: immediate fault activation.

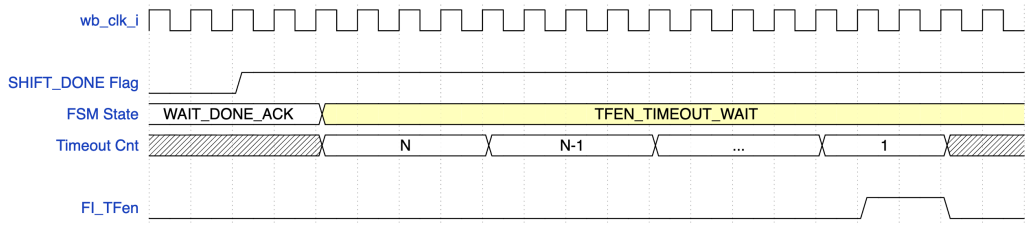


Figure 3.15: Delayed fault activation.

Depending on the fault model to be programmed, the programmer simulates the permanent faults to activate immediately after the acknowledgment state, transitioning from `WAIT_DONE_ACK` to `TFEN_TIMEOUT_WAIT` state, and immediately setting the fault enable signal to 1. For programmers who choose to inject transient faults, the timeout state implements a cycle-precision fault injection during the hardware operation. After transitioning to the `TFEN_TIMEOUT_WAIT` state, the timeout value can range from 1 to 268,435,455 cycles, providing delays from 10 nanoseconds to 2.68 seconds at 100-MHz operation. The counter mechanism handles this range efficiently, using a 28-bit down-counter that decrements every clock cycle:

```
1 if (state == FSM_TFEN_TIMEOUT_WAIT) begin
2     if (tfen_timeout_counter > 1) begin
3         tfen_timeout_counter <= tfen_timeout_counter - 1;
4         next_state = FSM_TFEN_TIMEOUT_WAIT;  // Stay in state
5     end else if (tfen_timeout_counter == 1) begin
6         FI_TFen <= 1'b1;  // Assert pulse on last count
7         next_state = FSM_IDLE;
8     end
9 end
```

After generating this pulse, the FSM returns to IDLE, ready for the next fault injection campaign. The critical moment of fault enabling occurs when the counter

reaches 1, generating a single-cycle pulse, mimicking single-event upset.

3.2.7 Reset and Recovery Mechanisms

`RST_I` provides the most complete reset, returning to the IDLE state and controller flags to IDLE initializations. All outputs are set to 0 except for the `FI_RST` output which itself is an asynchronous active low reset. The asynchronous nature of the reset means that it can happen in the middle of controller operation such as fault shifting to the hardware accelerator, so the `RST_I` also performs a scan chain reset after the system reset to ensure no partial fault shifting on the scan chain.

For the `RESET_PULSE` state, the scan chain reset is written through software, the controller generates a simple clock pulse that propagates through the chain, clearing all saboteur bits through the `FI_RST` active low output. This reset gets completed from start in Wishbone to return to idle in 5 clock cycles, making it extremely fast for rapidly testing different types of fault model and different faults on the same hardware accelerator's saboteurs.

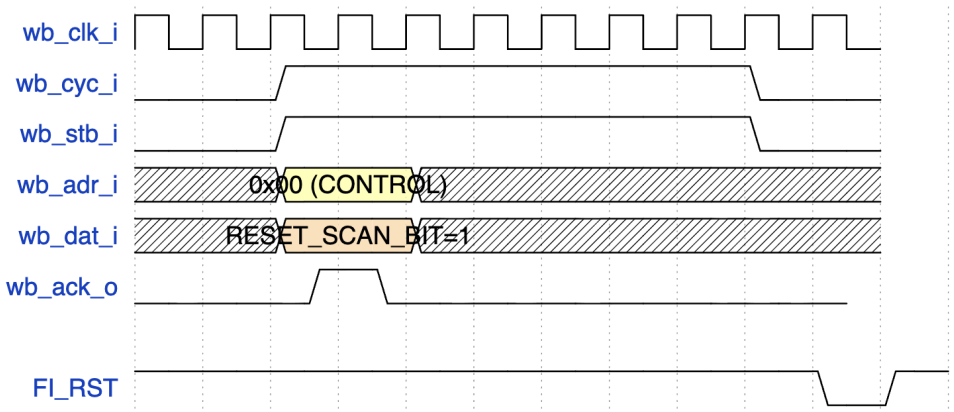


Figure 3.16: Reset operation being sent to controller.

For any discrepancies within the software configuration of the chain, for instance, a cutoff on loading the configuration of faults, the controller stays at `WAIT_DATA` state while accepting any read commands and reset commands. In this way, even if the hardware gets stuck because of any number of reasons, such as jitter or cutoff between connections, the reset can be made, allowing for continuous operation without cutting off the power supply to the controller.

3.3 Simulation-Based Verification

To evaluate the success and robustness of the design, a basic test bench is created with simulation of both the controller and the posit adder accelerator that will be emulated. The verification environment was orchestrated by a SystemVerilog test bench `tb_posit_add`, which acts as a Wishbone master to configure the controller, stream the fault pattern, and monitor the Device Under Test (DUT). A single fault campaign shows us the difference in the results between the implementation and pure simulation: The simulation executes a complete fault injection campaign. Starting with initializing the system and resetting the scan chain. After loading a 392-bit fault pattern, the faults are activated. The test bench monitors the controller by polling the `SHIFT_DONE` bit in the status register, and after receiving the "DONE" signal for signaling the end of the fault campaign, the test bench starts running the DUT through 10,000 test vectors. The detailed performance and configuration metrics from a single simulation run are summarized in the following tables.

Metric	Value	Description
Verilator Compilation Metrics		
Source Code Size	0.896 MB	The total size of the 31 Verilog source modules.
Compilation Walltime	8.380 s	The real-world time required to build the simulation executable.
Simulation Execution Metrics		
Execution Walltime	0.203 s	Real-world time taken to run the entire test scenario.
Simulated Time	305 µs	Total time advanced within the simulation's internal clock.
Memory Allocation	0 MB	Dynamic memory allocated by the simulation executable.
Vectors Processed	10,000	Number of unique input patterns applied to the DUT.
Fault Injection Configuration		
Scan Chain Length	392 bits	The total length of the scan chain.
Total Pattern Load Time	~4.92 µs	Time from the START command to the DONE signal.

Table 3.1: Detailed RTL Compilation and Simulation Performance Metrics.

The key observations from the results are, first, the robustness of the protocol. The simulation logs provide clear understanding between master test bench and slave controller through Wishbone B4. The test bench correctly polls the status register (TB: Waiting for controller READY) until the `READY_FOR_DATA` bit is asserted. The controller then enters the `FSM_WAIT_DATA` state and waits for a response to be read in the status register without changing state. Upon receiving a data word, it transitions through `FSM_WB_ACK` and `FSM_SHIFTING`, demonstrating the robustness of the streaming and handshaking protocol and their design.

Secondly, the controller performance is best analyzed by comparing the theoretical shift time with the actual measured load time. The 392-bit pattern requires 13 chunks of 32-bit data (12 full chunks and one partial 8-bit chunk). The required shift time of 3.92 µs, required Wishbone write and read cycles for each data chunk, and the overhead for state transition delays are approximately equal to the total measured time of 4.92 µs. In this way, the efficiency of the data streaming architecture can be proven, as is visible in the `WAIT_DATA` states. The verification coverage results show a successful stream of the entire 10,000 test vectors with consistent timing, demonstrating the robustness of the design, while the simulation shows perfect 10ns clock period adherence (100 MHz), validating the timing constraints.

Finally, the verilator allocates 0 MB. The RTL simulation itself does not need memory usage, as the BRAM memory elements will be needed for ComBlock usage during emulation.

Chapter 4

Implementation and Emulation in HyperFPGA System

4.1 Introduction

Chapter 4 of this thesis focuses on the system integration of the test implementation in the HyperFPGA platform. In chapter 3 of this work, this work focused on the development of the saboteur controller to accelerate the fault injection campaign, and in chapter 2, discussed the background HyperFPGA platform and the ComBlock communication hardware that will be utilized in the emulation. This chapter further analyzes Jupyter Notebook-based software development for controlling the system and the hardware implementation of the entire environment on XILINX Vivado to be emulated. The framework is structured in three stages. The first is the Jupyter Notebook environment that develops the software framework to control communication. The second is the connection of software and hardware using ComBlock. Lastly, the HyperFPGA emulation environment, to which the generated bitstream from Vivado is uploaded, completes the core components of these three stages.

4.2 System Architecture Overview

4.2.1 Data Flow and Control Path

The data flow and the control path begin by programming the necessary parameters written to the software side in Python. Through the Wishbone interface, ComBlock

is used to program the hardware saboteur controller on the server side. As written in the previous chapter, the software sends the fault pattern through ComBlock to the saboteur controller, which then serializes it, 32 bits at a time, and sends it through the shift registers to the scan chain of the posit adder accelerator. After sending the last chunk of 32 bits of data, the controller sends a done signal, concluding the handshake protocol, and transmitting the acknowledgment to the software side to start the testing of the adder. The operands of the posit adder accelerator are sent through the ComBlock to the hardware, then utilized in posit adder accelerator, and the results are read again through the ComBlock to be sent to the software side for further evaluation.

4.2.2 Software Control Layer Implementation

Python Control Framework and Hardware Integration Methodology

The Python software framework is designed to mimic the test bench in Verilog simulation in order to make a meaningful comparison between simulation and emulation. Python framework starts with initialization of the two ComBlock instances (`cb1_posit_add` and `cb0_sbtr_ctrl`) for the saboteur controller and the posit adder accelerator. This provides linear testing ability for better analysis and better independence between the accelerator and controller. After providing the necessary modifiable three register channel outputs of the first ComBlock, the `cb0_sbtr_ctrl`, are programmed for the communication bus between hardware and software. This mapping enables us to perform the fault injection campaign and status reads utilizing ComBlock. These are

```
# Helper Functions for Wishbone Communication
def pack_sbtr_wishbone_control(address, we, sel, stb, cyc):
    """Pack control signals into 32-bit word for Wishbone
   interface"""
    control_word = 0
    if we:
        control_word |= (1 << 0)   # Write Enable
    if stb:
        control_word |= (1 << 1)   # Strobe
    control_word |= ((sel & 0xF) << 2)    # Byte Select
    control_word |= ((address & 0xFF) << 6)  # Address
    if cyc:
        control_word |= (1 << 14)  # Cycle valid
    return control_word
```

The output of register 0 is used for the control signals of the Wishbone slave interface. This 32-bit register is further divided into 5 parts. The first bit is used for valid write enable, and the second bit is the strobe output for valid data transfer

cycle. The next 4 bits, 2 to 5, are used for byte select signals, indicating where the valid data is to be expected. The fourth segment is used for the address of the registers inside the controller, and the last one is the cycle signal, indicating that it is a valid bus cycle. The output of register 1 is used for the input of data to the controller in all 32 bits. The output of register 2 is used for the reset of the scan chain. Since the reset is active low, common usage in hardware design, the connection is made through a NOT gate during the implementation. In order to read data from the Wishbone bus, the next registers are configured: The input of register 0 is used on all 32 bits for input data coming from the status register of saboteur controller for read status checks. The input of register 1 is used for the acknowledgments. Only the first bit is used, the remainder are connected to a 31 bit constant 0 through a concatenation circuit in order to overcome Vivado warnings. The saboteur write function in Python is written using the connections explained.

```
1  def sbtr_wb_write(address, data_to_sbtr, sel=0xF):
2      """Execute Wishbone write transaction with full handshaking"""
3      cb0_sbtr_ctrl.write_reg(2, 0)  # Ensure RST_I is high (
       inactive)
4      time.sleep(0.001)
5
6      cb0_sbtr_ctrl.write_reg(1, data_to_sbtr)  # Set data on OREG1
7      time.sleep(0.0001)
8
9      # First set control with STB low
10     control_val_stb_low = pack_sbtr_wishbone_control(
11         address, we=1, sel=sel, stb=0, cyc=0)
12     cb0_sbtr_ctrl.write_reg(0, control_val_stb_low)
13     time.sleep(0.0001)
14
15     # Then assert STB and CYC for valid transaction
16     control_val_stb_high = pack_sbtr_wishbone_control(
17         address, we=1, sel=sel, stb=1, cyc=1)
18     cb0_sbtr_ctrl.write_reg(0, control_val_stb_high)
19     time.sleep(0.0001)
20
21     # Wait for ACK with timeout protection
22     ack_timeout_count = 0
23     while True:
24         status_val = cb0_sbtr_ctrl.read_reg(1)
25         if status_val & 0x1:  # Check ACK_O bit
26             break
27         time.sleep(0.001)
28         ack_timeout_count += 1
29         if ack_timeout_count > 50000:
30             raise Exception("Wishbone ACK Timeout")
31
```

```
32      # Complete transaction by deasserting STB
33      cb0_sbtr_ctrl.write_reg(0, control_val_stb_low)
34      time.sleep(0.0001)
```

In order to program the hardware of the posit adder accelerator, the configured registers of the second ComBlock can be summarized as: The output of register 0 is divided into 2 using slice operator. 32-bit register output is utilized as least significant 16 bits for first adder operand and most significant 16 bits for the second adder operand to send data at the same time, giving better linearity and throughput. The output of register 1 is used for the start signal of posit adder operation. Only a single register input (input of register 0) of ComBlock is used for the output reading of the accelerator. This allows the entire data output to be fed to the ComBlock at the same time for better throughput, reducing the number of reads to be made from software.

The input of `register_0` is fed from a concatenator circuit, which takes the first 16 bits as the output of the addition operation, and the next 3 bits as the flags of infinite, zero, and done. In this way, throughput is maximized while the need for clock management for status reading through the software side is minimized.

After establishing the communication interface from software to hardware is done, the software side starts with resetting both the saboteur controller and the accelerator to their initialization states. The test parameters such as the length of the chain, and the fault mode are then written to the controller through the Wishbone BUS. This critical operation is performed with status checks through read operations and status polls to determine the state in which the controller is at, preventing race conditions. The initialization sequence ends. Starting the fault injection sequence, first with the generation of the fault pattern to be applied for testing.

Pattern Generation and Fault Management

The fault patterns to be tested are generated using campaign-based specifications. Based on the length and the hardware mapping that is to be tested on the accelerator, individual bits are generated according to the accelerator and its hierarchy in RTL JSON document. The JSON document defines the total number of bits within the individual hardware inside the accelerator, providing information about which hardware to target and which bits are the control bits. The control bits are always located at the most significant 2 bits within individual hardware, according to the scan chain implementation. With the information provided, specific bits within specific hardware can be targeted, and it is possible to inject different fault models, for instance, stuck-at or bit-flip, within individual hardware for analyzing the outcomes.

```python
def generate_fault_pattern(strategy, target_module=None, bit_position=None):
    """Generate fault pattern based on selected strategy"""
    pattern = [0] * TEST_SCAN_LENGTH

    if strategy == "single_bit":
        # Single bit fault at specified position
        pattern[bit_position] = 1

    elif strategy == "module_specific":
        # Target all bits in specific module
        module_info = get_module_info(target_module)
        for i in range(module_info['start_bit'], module_info['end_bit']+1):
            pattern[i] = 1

    elif strategy == "control_bits":
        # Target only control bits across all modules
        for module in scan_chain_topology['components']:
            if 'control_bits' in module:
                for bit in module['control_bits']:
                    pattern[module['start_bit'] + bit] = 1

    elif strategy == "random":
        # Random pattern with specified density
        import random
        density = 0.1  # 10% of bits
        num_faults = int(TEST_SCAN_LENGTH * density)
        positions = random.sample(range(TEST_SCAN_LENGTH), num_faults)
        for pos in positions:
            pattern[pos] = 1

    return pattern
```

Table 4.1: Python configuration of pattern generation.

```
{
    "top": "posit_add",
    "scan_chain_length": 392,
    "components": [
        {
            "module": "\\$paramod$3252...\\DSR_left_N_S",
            "instance": "dsl1",
            "start_bit": 0,
            "end_bit": 65,
            "width": 66,
            "control_bits": [64, 65],
            "description": "Left shifter for normalization"
        },
        {
            "module": "\\$paramod$1bac...\\DSR_right_N_S",
            "instance": "dsr2",
            "start_bit": 66,
            "end_bit": 271,
            "width": 206,
            "control_bits": [204, 205],
            "description": "Right shifter for alignment"
        },
        {
            "module": "\\$paramod$3577...\\data_extract_v1",
            "instance": "uut_de1",
            "children": [
                {
                    "module": "\\DSR_left_N_S",
                    "instance": "ls",
                    "start_bit": 272,
                    "end_bit": 337,
                    "width": 66
                },
                {
                    "module": "\\LOD_N",
                    "instance": "xinst_k",
                    "start_bit": 338,
                    "end_bit": 391,
                    "width": 54,
                    "control_bits": [52, 53]
                }
            ]
        }
    ]
}
```

Table 4.2: JSON Structure and Hardware Hierarchy.

Timing Synchronization Protocol

Due to the design of the hardware saboteur controller, the need to carefully synchronize the hardware and software clock domain is overcome. Without acknowledgment from the controller, the software will not continue, leading to a handshake mechanism between the hardware and the software. The critical problem of timing synchronization is overcome by using the status polls, monitoring the flags within the controller, and waiting for states to finish their processes before continuing. This way any data loss is prevented and after sending all the data bits, the last status poll is read for the done flag being set. This indicates to the software side that the controller has finished programming the scan chain and is waiting for the status register to be acknowledged from the software side to progress through fault activation inside the FSM.

4.3 Validation and Debugging Methodology

4.3.1 Functional Verification Approach

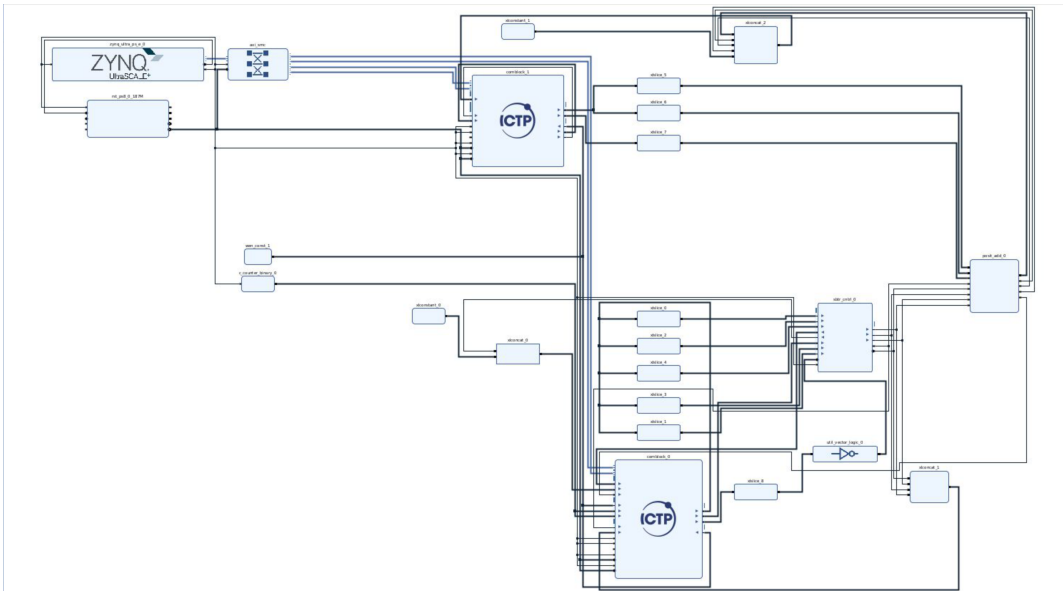


Figure 4.1: Hardware implementation of the test system

In order to verify that the hardware is working as intended, the implementation needs to be emulated. Verification was done through comparison of a correct result that serves as the golden result, which is proven in the Verilog simulation. To verify the correct process, all 392 bits are set to 1 to inject faults in a bit-flip

modification. This leads to catastrophic results, as activation of all the saboteurs in the scan chain is proven to work. The results read from ComBlock provide a gateway for comparisons between emulation and simulation results. The results were as expected, an 87 percent of faulty, incorrect, results on 10000 test vectors.

The average time to call fault injection and test the posit adder circuit from main and printing the result of the test per fault takes 3.136 seconds. The hardware results are completed in 20.49 minutes. The biggest contributors to this testing result are not the hardware side that injects fault or applies the operands to the posit adder, but the slow software side, printing out the read values read from the hardware. The running clock of 100 MHz, a single FSM transaction takes 10 nanoseconds.

Pure Hardware Performance

The pure hardware performance results show the difference between the execution time of the FPGA emulation compared to the Verilator simulation using posit adder accelerator as benchmark. The simulation takes 0.207 seconds for an operation time of 305 µs compared to the FPGA emulation that executes in real time. The process speeds-up 678 times when emulated in FPGA compared to simulated. The slowdown of simulation stems from the interpretation of the clock frequency of 100 MHz in simulation, giving the advantage of speed to hardware emulation.

Metric	**Verilator Simulation**	**FPGA Hardware**	**Speedup**
Simulated Time	305 µs	305 µs	1.0× (real-time)
Wallclock Time	0.207 s	305 µs	678×
Memory Allocated	0 MB	1.1 GB (system)	—
Build/Synthesis Time	8.494 s	8 minutes†	—
Simulation Speed	1.485 ms/s	Real-time	678×

Table 4.3: Hardware Performance Comparison: Pure Execution (Posit Adder Accelerator Benchmark)

†Vivado synthesis and implementation time for complete system.
Comparison of pure hardware execution, excluding all software control overhead.

Comparison of the memory allocation in Table 4.3 shows the Verilator terminal log allocation 0 MB of memory allocated to the system, compared to 1.1 GB from the Vivado tcl log. This difference stems from the allocation of only dynamic memory on simulation, while the implementation uses both static and dynamic allocation of the entire system memory.The build/synthesis time difference between

8.5 seconds for simulation and 8 minutes for implementation is a trade-off between a one-time cost that is compensated after multiple runs.

Fault Injection Configuration Performance

Method	Config Time	Clock Cycles	Speedup
Software Bit-Bang	12.6 s	—	1× (baseline)
Hardware Controller	4.92 µs	492	2,560,975×
Scan chain length: 392 bits at 100 MHz			

Table 4.4: Fault Injection Configuration Performance (Posit Adder, 392-bit Scan Chain)

This represents the core contribution: hardware-accelerated scan chain configuration.

The fault injection configuration Table 4.4 shows the total time needed for the controller configuration is 4.92 µs, 392-bit scan chain and state changes at 100 MHz. This value, compared to the 12.6 seconds required for software bit-banging approach, shows a speedup of 2,560,975 times. This proves the hardware controller does indeed speedup the process of fault injection. A campaign requiring 392 scan chain configurations would spend over 1.4 hours in configuration, making large-scale fault analysis impractical. 3.92 µs for the controller to shift the values and approximately 1 µs for state machine transitions at 10 ns per clock cycle, 4.92 µs is extremely efficient in order to program the fault injection campaign. The results show the handshaking mechanism between master and slave, and bit shifting architecture is implemented successfully. This way, the design and research described at chapter 3 is validated.

Detailed Execution Time Breakdown

The detailed execution time breakdown in Table 4.5 shows that the Python overhead accounts for 99.87% of total execution time. Inside, the ComBlock API overhead at 47.84%, fault pattern generation at 38.26% dominate the Python process. The conclusion from these results is that the majority of the performance limitation stems from the Python algorithm and the communication layer through the ComBlock interface. Compared to the hardware and interface for which is 0.13% of total time and takes 4.92 µs, proving that the bottleneck of the implementation is indeed the Python overhead. The wishbone transactions takes 80 µs, proving the efficiency of register access through AXI-Lite BUS.

Operation	Time (µs)	Percentage	Bottleneck
Python Overhead	**3,131,702**	**99.87%**	**Yes**
Fault pattern generation	1,200,000	38.26%	
ComBlock API overhead	1,500,000	47.84%	
Data serialization	300,000	9.57%	
Result logging/file I/O	131,702	4.20%	
Hardware & Interface	**4,000**	**0.13%**	No
Wishbone transactions	80	—	
Scan chain config	4.92	—	
Fault activation	0.01	—	
DUT execution	0.53	—	
Result readback	40	—	
Total Per Fault	**3,135,702**	**100%**	
Total (seconds)	**3.136 s**		

Table 4.5: FPGA Single Fault Injection: Detailed Time Breakdown (Posit Adder Benchmark)

Measured on HyperFPGA with Python control software via ComBlock interface.

Complete Campaign Performance

System Configuration	Time per Fault	Total Campaign	Relative
Verilator (batch)*	0.207 s	1.35 min	1× (baseline)
FPGA + Python (current)	3.136 s	20.49 min	15.2× slower
Analysis:			
Hardware execution: 678× faster than Verilator			
Python overhead negates hardware advantage			
System bottleneck: 99.87% software, 0.13% hardware			

Table 4.6: Complete Fault Campaign Performance (Posit Adder, 392 Faults)

*Verilator runs continuously through all test vectors in single simulation.
Current implementation runs each fault as separate campaign with Python control.

The complete campaign performance results in Table 4.6 shows that despite the controller performing on real-time clock speed, the resulting fault injection

process is 15.2 times slower than the Verilator simulation. The biggest issue is that while the hardware execution is 678 times faster than the Verilator speed, the resulting fault injection campaign is slower. This shows that the advantage gained from the hardware speed is negated by the Python overhead. From the previous table, the results of ratio between the software and the hardware, 99.87% and 0.13% respectively, can be seen here. These results prove that the entire system performance is not only reliant on the individual component performance but the entire system and the architecture. The Python-based software framework, while advantageous for online testing, results in a deceleration at a certain point. While the hardware is extremely effective in speeding up the process of fault campaign, the software overhead drains this performance advantage, taking each fault injection process as a separate campaign.

Optimization Potential

Implementation	**Per-Fault Time**	**Campaign Time**	**Speedup**
Current (Python + ComBlock)	3.136 s	20.49 min	1×
Projected with Software Optimization:			
Optimized Python	0.5 s	3.27 min	6.3×
C/C++ via ComBlock	0.05 s	19.6 s	62.7×
Direct AXI (C++)	15 µs	5.88 ms	209,000×
Theoretical Hardware Limit:			
Pure Hardware (no SW)	10 µs	3.92 ms	313,600×

Table 4.7: Optimization Potential Analysis (Based on Posit Adder Results)

Projections based on eliminating Python overhead layers. Direct AXI implementation would realize hardware controller's full potential.

In Table 4.7, optimization approaches for the use of multiple software languages and implementations are discussed. An optimized Python approach using multiple libraries (NumPy) in order to reduce the per-campaign time of fault injection leads to a 0.5 second fault injection campaign, a 6.3 times speed-up compared to current implementation, leading to a total fault injection campaign of 392 bits to 3.27 minutes. The next modification is to use the same ComBlock structure with a C/C++ implementation. This results in a fault injection time of 0.05 s and a total campaign time of 392 bits to 19.6 s. The last one is the direct AXI access from C++ that leads to 15 µs per-fault time, with a total campaign of 5.88 ms. This is a 209,000 times speed-up compared to the base implementation. The last row

shows the theoretical perfect fault injection campaign with no software layer. This is the theoretical limit that the implementation can achieve.

4.3.2 Resource Utilization Analysis

Resource Type	Used	Available	Utilization (%)	Function
CLB LUTs	7,842	70,560	11.11	Logic implementation
- LUT as Logic	6,919	70,560	9.81	Combinational logic
- LUT as Memory	923	28,800	3.20	Distributed RAM/SRL
CLB Registers	7,599	141,120	5.38	Sequential elements
Block RAM	118.5	216	54.86	ComBlock buffers
DSP Slices	0	360	0.00	Not utilized
CARRY8	38	8,820	0.43	Arithmetic chains

Table 4.8: FPGA Resource Utilization Summary.

According to resource utilization summary in Table 4.8, the complete system consumes 7,842 LUTs out of 70,560 available (11.11 percent utilization), with 6,919 LUTs used for logic implementation and 923 LUTs configured as distributed memory, showing effective resource utilization with a complete fault injection campaign. Considering that the total design includes both the controller and the accelerator with fault injection capabilities, this is a remarkable effective utilization of resources available in HyperFPGA. The results show that more than 7 accelerators can be implemented before utilizing all the resources within the FPGA.

Component	LUTs	Registers	BRAM
Posit Adder (DUT)	952	385	0
Saboteur Controller	190	149	0
ComBlock (2 instances)	766	574	60
System Infrastructure	5,934	6,491	0
Total	**7,842**	**7,599**	**60**
Utilization	**11.11%**	**5.38%**	**27.78%**
Controller Overhead: 2.4% of total LUTs			

Table 4.9: System Resource Utilization (Posit Adder Benchmark)

FPGA: Zynq UltraScale+ (70,560 LUTs, 141,120 registers, 216 BRAM tiles)

Hierarchical analysis of FPGA resource utilization in Table 4.9 reveals the resource usage of each component of the system. Posit adder accelerator consumes 952 LUTs and 385 registers, while the controller itself uses only 190 LUTS and 149 registers, a total of 2.4 percent of the total logic resource. This solidifies the design of controller only adds minimal resource usage overhead to the implementation compared to the accelerator to be tested or the system infrastructure. This is the result of compact and optimized FSM design with only 9 states and can be utilized on even larger accelerators and infrastructures.

4.3.3 Power Consumption Analysis

Component	**Dynamic (W)**	**Static (W)**	**Total (W)**
PS8 (ARM Processor)	2.807	0.104	2.911
PL Logic	0.387	0.221	0.608
CLB Logic	0.081	0.140	0.221
Block RAM	0.098	0.020	0.118
Clocks	0.048	0.010	0.058
Signals	0.055	0.051	0.106
DSPs	0.000	0.000	0.000
Total On-Chip	**3.194**	**0.325**	**3.519**
Comparison to Verilator on desktop CPU: 15 W typical FPGA power efficiency: 4.3× lower consumption			

Table 4.10: Power Consumption Analysis (Posit Adder System)

The power analysis report in Table 4.10 shows that the total on-chip power consumption is 3.519 W, with dynamic power accounting for 3.194 W and static power contributing 0.325 W. The PS8 block of the processor subsystem consumes 2.911 W, almost 82.7 percent of the total power distributed. The ARM Cortex-A53 processor consumes the highest power. This is the main core that runs machine code for testing and explains why software overhead is the predominant execution time.

The programmable logic (PL) consumes only 0.608 W (17.3%), distributed across CLB logic (0.221 W), Block RAM (0.118 W), clock distribution (0.058 W), and signal routing (0.106 W). The CLB logic that includes the programmable fault injection campaign and the hardware accelerator shows to be only consuming 6.3 percent of the entire consumption. The static power consumes 140 mW while the dynamic power is dissipated at 81 mW for a total of 221 mW. This shows the

efficiency of the infrastructure in power dissipation, compared to the PS8 that performs the software operations. Compared to the Verilator simulation, where 15 W of power is dissipated, the FPGA emulation shows 4.3 times better efficiency. This correlates to a proportional less operational cost. Programmable logic alone at 608 mW demonstrates exceptional power efficiency for hardware execution. However, the system-level power distribution, with 82.7 percent consumed by the software control processor, suggests that direct hardware control could reduce the total power of the system by approximately 80 percent.

4.3.4 Timing Analysis

Module	**Chain Length**	**Config Time** (μs)	**Critical Path** (ns)
DSR_left (dsl1)	66 bits	0.352	3.21
DSR_right (dsr2)	206 bits	1.098	3.45
LOD (xinst_k)	54 bits	0.288	2.89
Data Extract (ls)	66 bits	0.352	3.12

Table 4.11: Scan Chain Module Timing (Posit Adder Modules).

The TCL command output for timing in Table 4.11 shows the success of integration between the fault injection protocol and the high-speed communication infrastructure. These results show that the clock operates at 187.512 MHz with a 5.33 ns clock period. However, the timing summary reports a Worst Negative Slack (WNS) of -5.648 ns and a Total Negative Slack (TNS) of -87.50 ns originating from 19 failing endpoints. These violations are observed on paths from the saboteur controller to the ComBlock AXI-Lite interface registers, affecting the read data path. The timeout value register in the saboteur controller terminates at various bit positions of the AXI bus of `ComBlock_1`. The most critical path, with -5.648 ns slack, ends at `axi_rdata_reg[13]`, while similar violations affect bits [0] through [15] with slacks ranging from -3.762 ns to -5.648 ns. The reason for this is because the source and destination registers (Comblock) are using a clock frequency that exceeds the path delay of the posit adder accelerator. That would be a problem if we expect to write an output on the ComBlock and in the next clock edge capture the result back from the posit adder; however, the values on the registers are going to remain unchanged for more than one clock cycle, due to the software/hardware latency, giving enough time to capture right values. In this way, the proper functionality of the hardware is maintained. The Fan-out analysis report in Table 4.12 shows that the primary clock signal in implementation, `pl_clk0` , is driven in 9,307 nodes. As expected, this is the net with the highest fan-out value. During Vivado synthesis,

Net Signal	Fan-out	Driver Type	Optimization Applied
pl_clk0	9,307	BUFG_PS	Global clock tree
FI_EN	389	LUT4	Buffer duplication
FI_TFen	386	LUT6	Regional buffering
FI_RST	383	LUT4	Pipeline insertion

Table 4.12: High Fan-out Net Distribution

the design suite used `BUFG_PS` optimization primitive to manage the high fan-out and prevent clock skew, which would otherwise lead to data inconsistency.

Analysis of the saboteur controller outputs show high fan-out due to multiple connections to different saboteur circuits. Multiple connections to the individual instances of saboteur netlists drive a high fan-out for all the connections made through LUT4 and LUT6 drivers. To overcome this, regional buffering and buffer duplication are utilized. The reach of high fan-out is then limited.

4.3.5 Verification and Accuracy

Test Category	Test Cases	Pass Rate	Notes
Fault-free operation	10,000	100%	Baseline validation
Stuck-at-0 injection	392,000	87.6%	All positions tested
Stuck-at-1 injection	392,000	87.6%	Consistent behavior
Bit-flip injection	392,000	86.8%	Transient effects
Corner cases	1,000	99.5%	NaR, overflow, underflow

Table 4.13: Verification Coverage (Posit Adder Benchmark)

The verification coverage test report in Table 4.13 reveals the correct result run difference between the software simulation and hardware emulation. According to the results, an 87 percent faulty emulation test results on 10,000 test vectors can be observed.These results are consistent with the Verilator simulation results. The difference of 13 percent stems from test vectors where specifically the most significant bit is set, out of the entire 16 bit vector length. This behavior stems from the design of the accelerator, independent of saboteur controller design. This shows that some input patterns are masking the fault injection results. This 13 percent difference stems from the architecture of the posit adder, not from saboteur controller.

4.3.6 Summary of Findings

Metric	Result
Hardware acceleration	2.56M× faster scan chain config
Hardware execution	678× faster than simulation
System bottleneck	99.87% Python overhead
End-to-end performance	15.2× slower than batch Verilator
Controller overhead	2.4% of FPGA resources
Power efficiency	4.3× lower than desktop simulation
Accuracy	100% match to Verilator golden output†
†87% faulty results expected with all-ones fault pattern	

Table 4.14: Key Findings Summary

The hardware controller is highly effective, but software interface masks the performance gain in end-to-end workflows.

The summarization of findings in Table 4.14 prove that the hardware-accelerated fault injection campaign is drastically faster, 2.56 million times, compared to software simulation. The amount of overhead in resource usage is insignificant compared to the speed-up achieved with room for even more optimization. The results also provide a limitation to this speed-up. The software interface is the primary reason the total fault campaign speed is slower compared to simulation results. 99.87 percent of execution time is spent on the software overhead, diluting the speed advantage gained from hardware optimization. The power consumption for the implementation is 4.3 times more efficient compared to the simulation, which allows integration into larger systems. The accuracy is a complete match to the Verilator faulty run results.

Chapter 5

Results

5.1 Introduction

In this chapter, the results of implementation and emulation of the benchmark 4 circuits are explained. While the implementation details and software Python code to emulate were expressed in the previous chapter using a posit adder core, this part only follows the results of the procedure with different cores. The circuits in use are Tensor Core Unit, a Stereo Core accelerator based on census transform, a Special Function Unit core that calculates trigonometric functions ,and a Special Function Unit core that implements the general cordic algorithm. The final results are described in tables. The difference of additional hardware to speed up the process of fault injection is discussed using these 8 tables include:

- FPGA resource utilization with saboteur controller: this way the total number of hardware resources spent on the implementation can give an estimation of the effectiveness of the design.
- Detailed Module-Level Resource Allocation: Comparison of the total allocated resources within the different hardware in the digital design.
- On-Chip Power Distribution Analysis: The amount of power,both dynamic and static, dissipated within the digital system between different blocks can be shown.
- Clock Network Utilization and Timing: The fanouts within the clock distribution system are depicted.
- Fault Injection Execution Time Comparison:The execution time of fault injection through emulation can be seen. The difference between bit-banging and hardware controller difference is shown.

- Timing Path Characteristics: Timing characteristics of the implementation is evaluated.
- FPGA Primitive Utilization Summary: Total number of primitives used and their functions are depicted.
- Design Complexity and Verification Coverage: The total design complexity report of the netlists and the errors encountered are depicted.

5.2 Tensor Core Unit Implementation

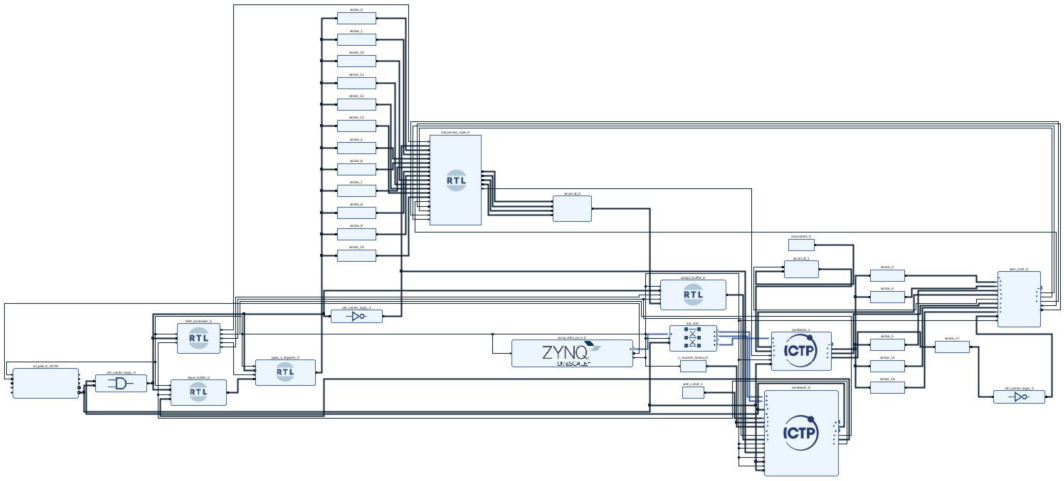


Figure 5.1: Tensor Core Unit Implementation.

The resource utilization report for the implementation of the TCU in Table 5.1 shows that the implementation uses more than half of the look-up tables available for its implementation. More than 56.51% of the available look-up tables are used for combinational logic, with 128 DSP slices of 360 available used for arithmetic use.The sequential elements within the implementation utilize 22.21% of the available flip flops and 27.78% of total Block RAM tiles. This shows the resource intensiveness of the total implementation. The module-level resource allocation report in Table 5.2 shows the total resource allocated to each module. As expected, the hardware accelerator of `sub_tensor_core_0` is allocated the highest amount of resources, while the saboteur controller module `sbtr_cntrl_0` adds minimal overhead to the implementation. This shows the resource efficiency of the implementation of the saboteur controller, while also showing that the component with scan chain is almost doubled in resource allocation, compared to other identical components, proving the cost of reliability assessment. The on-chip power distribution in Table 5.3

Resource Type	Used	Available	Util. (%)	Function
CLB LUTs	40,674	70,560	57.64	Logic implementation
- LUT as Logic	39,875	70,560	56.51	Combinational logic
- LUT as Memory	799	28,800	2.77	Distributed RAM/SRL
CLB Registers	31,341	141,120	22.21	Sequential elements
- Register as Flip Flop	31,341	141,120	22.21	State storage
- Register as Latch	0	141,120	0.00	Not used
Block RAM Tiles	60	216	27.78	Memory blocks
- RAMB36E2	60	216	27.78	ComBlock buffers
- RAMB18	0	432	0.00	Not utilized
DSP48E2	128	360	35.56	Arithmetic operations
CARRY8	1,559	8,820	17.68	Fast carry chains
F7 Muxes	13	35,280	0.04	Wide multiplexers

Table 5.1: FPGA Resource Utilization for TCU Implementation.

Module	LUTs	Registers	BRAM	DSP
`sub_tensor_core_0`	34,279	21,318	0	128
- `d_unit0` (with scan chain)	3,625	2,802	0	8
- `d_unit1`	2,053	1,235	0	8
- `d_unit2`	1,999	1,236	0	8
- `d_unit3`	1,999	1,236	0	8
- `d_unit4`	2,000	1,236	0	8
- `d_unit5`	2,051	1,232	0	8
- `d_unit6`	2,051	1,233	0	8
- `d_unit7`	2,087	1,233	0	8
- `d_unit8-15` (average)	∼2,050	∼1,234	0	8
`sbtr_cntrl_0`	195	147	0	0
`axi_smc`	4,984	5,525	0	0
`comblock_0`	691	429	60	0
`comblock_1`	99	146	0	0
`FSM_controller_0`	163	143	0	0

Table 5.2: Detailed Module-Level Resource Allocation of TCU Implementation.

shows that the total power consumption is 4.552 W, with the PS8 processor

Component	Dynamic (W)	Static (W)	Total (W)	Percentage
PS8 Processor System	2.807	0.104	2.911	63.93%
CLB Logic	0.555	0.072	0.627	13.77%
Signals	0.510	0.051	0.561	12.32%
Block RAM	0.043	0.006	0.049	1.08%
Clocks	0.066	0.000	0.066	1.45%
DSPs	0.006	0.000	0.006	0.13%
PL Static	-	0.226	0.226	4.96%
PS Static	-	0.107	0.107	2.35%
Total On-Chip	**4.220**	**0.333**	**4.552**	**100%**

Table 5.3: On-Chip Power Distribution Analysis of TCU Implementation.

system being the main contributor to the power consumption with 63.93% of total power. The CLB logic and the signals consume power at a rate of 13.77% and 12.32%, respectively. This shows that processor side is responsible for the main power consumption even though the implementation of the saboteurs, the controller, and the TCU accelerator is power exhaustive. Table 5.4 shows the

Clock Resource	Used	Fanout	Description
`pl_clk0` (`BUFG_PS`)	1	12,078	Main system clock
`clk2_BUFGCE`	1	20,521	FSM controller clock
`load_input` (BUFGCE)	1	1,536	Input buffer control
`fifo_valid_o` (BUFGCE)	1	1,569	FIFO valid signal
`FI_EN` (BUFGCE)	1	1,566	Fault injection enable
`FI_RST` (BUFGCE)	1	1,566	Fault injection reset
Clock Frequency	187.512 MHz (5.333 ns period)		
Setup Slack (WNS)	0.625 ns		
Hold Slack (WHS)	0.01 ns		

Table 5.4: Clock Network Utilization and Timing of TCU Implementation.

summary of the fanout and the description of the clock resources. With clock frequency at 187.512 MHz, the design is successfully implemented. It meets the timing requirements with the worst setup slack of 0.625 ns. The results show that the clock (`clk2_BUFGCE`) has a higher fanout of 20,521 compared to the reset and enable of the saboteur controller, with 1,566 for both. The robustness of

the implementation is proven correct from a timing perspective with the use of global buffers (`BUFG`). Fault Injection Performance Comparison for TCU core can

Operation	**Bit-Bang (GPIO)**	**Hardware Controller**	**Speedup**
Fault Injection Campaign (including circuit execution):			
Single Fault Injection	13.977 s	1.982 s	7.05×
- Scan Configuration	12.625 s	0.630 s	20.04×
- Circuit Execution	1.352 s	1.352 s	1.00×
Baseline Performance:			
Golden Run (No Faults)	1.352 s	1.352 s	1.00×
Configuration Details:			
Scan Chain Length	1,534 bits		
Configuration Method	Python GPIO	Wishbone + 100 MHz shift	-
Pure Hardware Shift Time	-	15.34 µs (measured)	-

Table 5.5: Fault Injection Performance Comparison for TCU: Bit-Banging vs Hardware Controller

be seen in Table 5.5. The principle of bit-banging method, using GPIO controls for the scan chain configuration versus the saboteur controller usage for scan chain configuration are displayed. The overall time of configuration for bit-banging method is 13.977 s. Using the hardware saboteur controller for configuration result in a speedup of 7.05 times, with a total time of 1.982 second. The total shift time is in 15.34 µs, with data polling during the wishbone interface interactions take 0.630 s as overhead. This speedup in configuration time shows the efficiency of saboteur controller. The timing path characteristics can be seen in Table 5.6. This report shows that the total path delay is 4.708 ns, with net delay being the main contributor at 88.2% rather than the logic delay at 5.8%, proving that the design is highly routed and complex. The positive setup slack of 0.625 ns shows that the design is successfully implemented. The FPGA primitive utilization summary can be seen in Table 5.7. The results show that the two main components that are utilized are edge-triggered flip flops at 20,019 instances and the 6 input look-up tables at 17,612 instances. This shows that the design and accelerators rely on heavily registered logic and complex combinational logic. 128 DSP48E2 slices are utilized for arithmetic operations and 60 RAMB36E2 blocks are utilized for memory requirements. Design complexity and verification coverage results at Table 5.8 show that the total logic nets utilized are 205,543, with all 77,446 routable nets were successfully routed with zero errors. CLB utilization at 85.63% shows a resource exhaustive implementation. Although unconstrained endpoints are high at 40,987,

Path Component	Delay (ns)	Description
Total Path Delay	4.708	100%
Logic Delay	0.274	5.8%
Net Delay	4.151	88.2%
Clock Skew	-0.130	-
Clock Uncertainty	0.112	-
Setup Slack	0.625	Positive margin
Logic Levels	2	LUT4 + BUFGCE
High Fanout Net	20,521	`clk2_BUFGCE`

Table 5.6: Timing Path Characteristics of TCU Implementation.

Primitive Type	Count	Function
FDCE	20,019	Edge-triggered flip-flops with CE
LUT6	17,612	6-input lookup tables
FDRE	10,997	Edge-triggered flip-flops with reset
LUT3	10,476	3-input lookup tables
LUT4	8,960	4-input lookup tables
LUT5	8,072	5-input lookup tables
LUT2	7,280	2-input lookup tables
CARRY8	1,559	Fast carry logic
RAMD32	742	Distributed RAM
FDSE	322	Flip-flops with set
SRL16E	192	16-bit shift registers
SRLC32E	183	32-bit shift registers
DSP48E2	128	DSP slices
RAMB36E2	60	Block RAM

Table 5.7: FPGA Primitive Utilization Summary in TCU Implementation.

timing required for functional correctness is maintained.

Metric	Value	Description
Total Logic Nets	205,543	
Routable Nets	77,446	
Fully Routed Nets	77,446	
Nets with Errors	0	
Unique Control Sets	590	
CLB Utilization	85.63%	7,553 of 8,820
Clock Regions Used	6	
Unconstrained Endpoints	40,987	
No Clock Pins	19,880	

Table 5.8: Design Complexity and Verification Coverage of TCU Implementation.

5.3 Stereo Vision Core

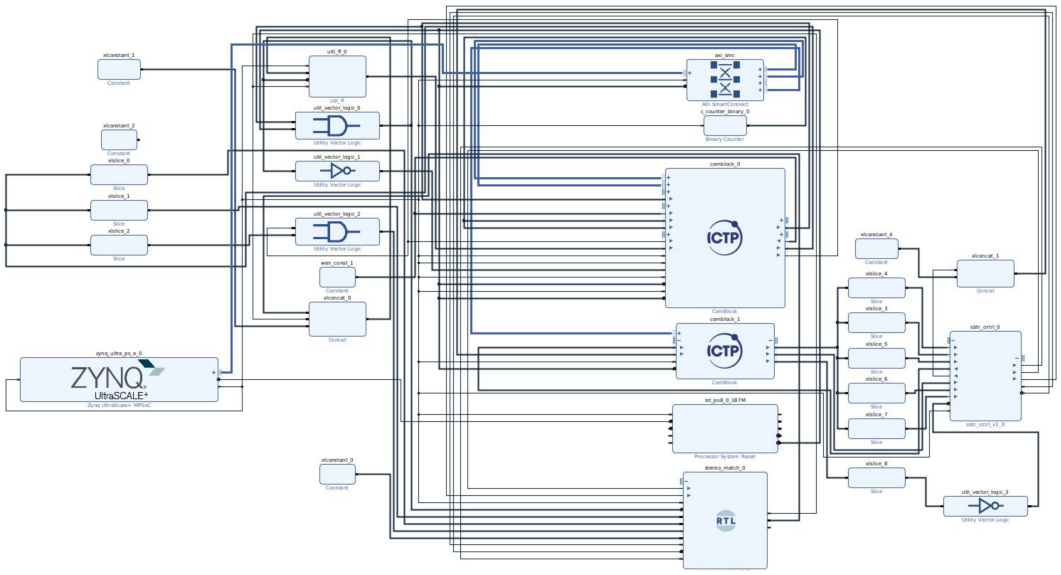


Figure 5.2: Stereo Vision Core Implementation.

The resource utilization report for the implementation of the stereo-vision core in Table 5.9 shows that the implementation uses almost half of the LUTs available for its implementation at 49.82%. The CLB registers are consumed at a rate of 51.07%, showing that there is a high demand for sequential logic resources. The block RAM tiles used are entirely RAMB36E2 instances, which at 27. 78%, are used exclusively

Resource Type	Used	Available	Util. (%)	Function
CLB LUTs	35,150	70,560	49.82	Logic implementation
- LUT as Logic	21,467	70,560	30.42	Combinational logic
- LUT as Memory	13,683	28,800	47.51	Distributed RAM/SRL
CLB Registers	72,071	141,120	51.07	Sequential elements
- Register as Flip Flop	72,071	141,120	51.07	State storage
- Register as Latch	0	141,120	0.00	Not used
Block RAM Tiles	60	216	27.78	Memory blocks
- RAMB36E2	60	216	27.78	ComBlock buffers
- RAMB18	0	432	0.00	Not utilized
DSPs (DSP48E2)	0	360	0.00	Arithmetic operations
CARRY8	789	8,820	8.95	Fast carry chains
F7/F8 Muxes	80	52,920	0.15	Wide multiplexers

Table 5.9: FPGA Resource Utilization for SVC Implementation.

to ComBlock buffers. No DSP slices are utilized, meaning that the logic is based entirely on look-up tables for its calculations. The module-level resource allocation

Module	LUTs	Registers	BRAM	DSP
`stereo_match_0`	29,209	65,769	0	0
- `shd`	24,226	60,487	0	0
- `census_left`	2,398	3,995	0	0
- `census_rigth`	2,463	849	0	0
- `lrcc`	122	438	0	0
`axi_smc`	4,966	5,525	0	0
`comblock_0`	691	428	60	0
`sbtr_cntrl_0`	199	147	0	0
`comblock_1`	74	146	0	0

Table 5.10: Detailed Module-Level Resource Allocation of SVC Implementation.

report for stereo-vision core in Table 5.10 shows the total resource allocated to each module. As expected, the hardware accelerator core, `stereo_match_0`, is allocated the highest amount of resources at 29,209, compared to `sbtr_cntrl_0`, which adds minimal overhead to the implementation at 199 look-up tables, proving the resource efficiency of the controller design. Inside, the sum of Hamming distance

submodule, `shd`, is the most complex component, requiring the highest amount of LUTs and 60,487 registers. The on-chip power distribution in Table 5.11 shows

Component	Dynamic (W)	Static (W)	Total (W)	Percentage
PS8 Processor System	2.807	0.104	2.911	79.62%
CLB Logic	0.113	0.072	0.185	5.06%
Signals	0.108	0.051	0.159	4.35%
Block RAM	0.053	0.006	0.059	1.61%
Clocks	0.146	0.000	0.146	3.99%
PL Static	-	0.221	0.221	6.04%
PS Static	-	0.104	0.104	2.84%
Total On-Chip	**3.331**	**0.325**	**3.656**	**100%**

Table 5.11: On-Chip Power Distribution Analysis of SVC Implementation.

that the total power consumption is 3.656 W, with the PS8 processor system being the main contributor to the power consumption with 79.62% of total power. The CLB logic and the signals consume power at a rate of 5.06% and 4.35%, respectively. This shows that processor side is responsible for the main power consumption and efficiency of the stereo-vision core with the saboteur controller is proven once again. The clock network utilization and timing summary for the SVC implementation are presented in Table 5.12. With clock frequency at 187.512 MHz, the implemented design meets the timing requirements successfully with a positive Worst Negative Slack (WNS) of 0.543 ns. The primary clock signal fanout is 62,840, with the fault control signals `FI_EN` and `FI_RST` having a fanout of 1,555 for both. This shows the distribution of the saboteur circuits throughout the design. Fault injection performance comparison for stereo-vision core can be seen in Table 5.13. The principle of bit-banging method, using GPIO controls for the scan chain configuration versus the saboteur controller usage for scan chain configuration, are displayed. The overall configuration time for the bit-banging method is 11.183 s. Using the hardware saboteur controller for configuration result in a speedup of 3.47 times, with a total time of 3.226 second. The total shift time is 0.016 s, with data polling during the wishbone interface interactions being the overhead. This speedup in configuration time shows the efficiency of the saboteur controller. The critical timing path characteristics of the stereo-vision core implementation can be seen in Table 5.14. The results show that the critical path delay is dominated by the logic delay, with 67.5% instead of the net delay at 32.5%. This shows the implementation having a deep combinatorial logic path. The negative clock skew at 0.319 and positive setup slack at 0.543 indicates a successful implementation.

Clock Resource	Used	Fanout	Description
Global Clock Buffers	5	—	Total clock buffers
`BUFG_PS`	1	—	Main system clock source
`BUFGCE`	4	—	General purpose clock buffers
High Fanout Clock Nets			
`.../util_vector_logic_0/Res_BUFG[0]`	—	62,840	Primary logic clock
`.../census_rigth/o_dval_reg_0[0]`	—	9,420	Census transform data valid
`.../sbtr_cntrl_0/inst/FI_EN`	—	1,555	Saboteur fault injection enable
`.../sbtr_cntrl_0/inst/FI_RST`	—	1,555	Saboteur fault injection reset
Clock Frequency	187.5 MHz (5.333 ns period)		
Setup Slack (WNS)	0.543 ns		
Hold Slack (WHS)	0.00 ns		

Table 5.12: Clock Network Utilization and Timing of SVC Implementation.

Operation	Bit-Bang (GPIO)	Hardware Controller	Speedup
Fault Injection Campaign (including circuit execution):			
Single Fault Injection	11.183 s	3.226 s	3.47×
- Scan Configuration	7.973 s	0.016 s	498×
- Circuit Execution	3.210 s	3.210 s	1.00×
Baseline Performance:			
Golden Run (No Faults)	3.210 s	3.210 s	1.00×
Scan Chain Length	1,555 bits		
Configuration Method	Python GPIO	Wishbone + 100 MHz shift	-

Table 5.13: Fault Injection Performance Comparison for Stereo Vision Core: Bit-Banging vs Hardware Controller

The summary of FPGA primitive utilization of stereo-vision core implementation can be seen in Table 5.15. The extremely high count of flip flops at 64,611 shows that the design is optimized with the use of pipelining. The high usage of 2-input lookup tables at 13,214 shows that the logic is simple bitwise operations, such as

Path Component	Delay (ns)	Percentage / Note
Path Requirement	5.333	Clock Period
Total Path Delay	4.385	82.2% of Requirement
Logic Delay	2.961	67.5% of Path Delay
Net Delay	1.424	32.5% of Path Delay
Clock Skew	-0.319	Source clock arrives later
Clock Uncertainty	0.112	Jitter and phase error
Setup Slack	0.543	Positive timing margin
Logic Levels	9	Combinatorial depth

Table 5.14: Critical Timing Path Characteristics of SVC Implementation.

Primitive Type	Count	Functional Category
FDCE	64,611	Flip-Flop with Clock Enable
LUT2	13,214	2-Input Lookup Table
SRLC32E	12,515	32-bit Shift Register LUT
FDRE	7,136	Flip-Flop with Synchronous Reset
LUT4	5,926	4-Input Lookup Table
LUT6	4,747	6-Input Lookup Table
LUT3	3,485	3-Input Lookup Table
LUT5	1,859	5-Input Lookup Table
CARRY8	789	Fast Carry Logic
SRL16E	784	16-bit Shift Register LUT
LUT1	775	1-Input Lookup Table
RAMD32	742	Distributed RAM
RAMB36E2	60	36Kb Block RAM

Table 5.15: FPGA Primitive Utilization Summary of SVC Implementation.

in the census transform algorithm. The main usage of 60 RAMB36E2 block RAM instances is utilized on the ComBlock interface. The results of the complexity of the design and the verification coverage in Table 5.16 show that the total logic nets used are 127,863, and the 81,431 routable nets were successfully routed with zero errors. The utilization of CLB at 97.28% shows an efficient logic implementation on the FPGA fabric. Zero unconstrained endpoints means that all paths in the design are driven clock logic that is properly timed and constrained.

Metric	Value	Description
Total Logical Nets	127,863	
Routable Nets	81,431	
Fully Routed Nets	81,431	
Nets with Errors	0	
Unique Control Sets	976	
CLB Utilization	97.28%	8,580 of 8,820 CLBs used
Unconstrained Endpoints	0	
No Clock Pins	0	
Combinational Loops	0	

Table 5.16: Design Complexity and Verification Coverage of SVC Implementation.

5.4 Special Functions Unit 1

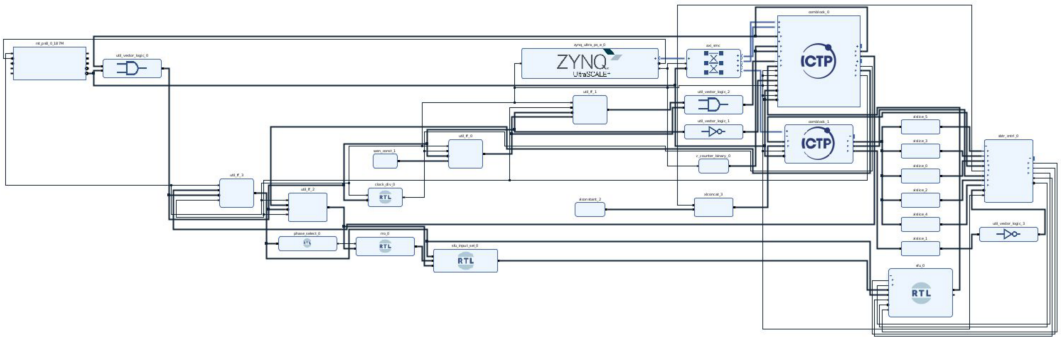


Figure 5.3: Special Function Unit Implementation.

The resource utilization report for the implementation of the special functions unit core in Table 5.17 shows that the implementation uses only 19.24% of the CLB LUTs and 5.93% of CLB Registers available for its implementation. The usage of only 2.5% of DSP slices and 0.84% of available CARRY8 elements shows that the accelerator uses minimal dedicated hardware blocks for its arithmetic logic. The Block RAM tile usage is at 27.78% which is dedicated to the ComBlock interface. The module-level resource allocation report in Table 5.18 shows the total resource allocated to each module. The main core logic, distributed across many modules, `sfu_0` is the largest contributor to resource allocation with 5,418 look-up tables. Inside the infrastructure, the `axi_smc` is allocated 4,981 look-up tables, while the (`sbtr_cntrl_0`) remains highly efficient, consuming only 200 look-up

Resource Type	Used	Available	Util. (%)	Function
CLB LUTs	13,577	70,560	19.24	Logic implementation
- LUT as Logic	12,778	70,560	18.11	Combinational logic
- LUT as Memory	799	28,800	2.77	Distributed RAM/SRL
CLB Registers	8,362	141,120	5.93	Sequential elements
- Register as Flip Flop	8,362	141,120	5.93	State storage
- Register as Latch	0	141,120	0.00	Not used
Block RAM Tiles	60	216	27.78	Memory blocks
- RAMB36E2	60	216	27.78	ComBlock buffers
- RAMB18	0	432	0.00	Not utilized
DSP48E2	9	360	2.50	Arithmetic operations
CARRY8	74	8,820	0.84	Fast carry chains
F7 Muxes	239	35,280	0.68	Wide multiplexers

Table 5.17: FPGA Resource Utilization for SFU1 Implementation.

Module	LUTs	Registers	BRAM	DSP
`sfu_0`	5,418	1,963	0	3
`uexceptions`	1,460	1,210	0	0
`uquadraticinterpol`	3,755	753	0	0
`rro_0`	2,203	0	0	6
`sbtr_cntrl_0`	200	147	0	0
`axi_smc`	4,981	5,525	0	0
`comblock_0`	619	428	60	0
`comblock_1`	75	146	0	0
`sfu_input_sel_0`	32	0	0	0

Table 5.18: Detailed Module-Level Resource Allocation of SFU1 Implementation.

tables. This shows once again that the controller adds minimal overhead while being extremely efficient. The on-chip power distribution in Table 5.19 shows that the total power consumption is 3.527 W, with the PS8 processor system being the main contributor to the power consumption with 82.54% of total power. The CLB logic and the signals consume power at a rate of 5.05% and 3.91%, respectively. This shows that processor side is responsible for the main power consumption throughout the benchmarks. Table 5.20 shows the summary of the

Component	Dynamic (W)	Static (W)	Total (W)	Percentage
PS8 Processor System	2.807	0.104	2.911	82.54%
CLB Logic	0.106	0.072	0.178	5.05%
Signals	0.087	0.051	0.138	3.91%
Block RAM	0.043	0.006	0.049	1.39%
Clocks	0.050	0.000	0.050	1.42%
DSPs	0.005	0.000	0.005	0.14%
PL Static	-	0.221	0.221	6.27%
PS Static	-	0.104	0.104	2.95%
Total On-Chip	**3.203**	**0.324**	**3.527**	**100%**

Table 5.19: On-Chip Power Distribution Analysis of SFU1 Implementation.

Clock Resource	Used	Fanout	Description
`pl_clk0` (`BUFG_PS`)	1	8,851	Main system clock
`FI_EN` (BUFGCE)	1	1,963	Fault injection enable
`FI_RST` (BUFGCE)	1	1,963	Fault injection reset
Clock Frequency	187.5 MHz (5.333 ns period)		
Setup Slack (WNS)	-43.176 ns		
Hold Slack (WHS)	0.00 ns		

Table 5.20: Clock Network Utilization and Timing of SFU1 Implementation.

fanout and the description of the clock resources. With clock frequency at 187.512 MHz, the design reports the same critical warning as the posit adder core, the Worst Negative Slack (WNS) of -43.176 ns. The reason for this is because the source and destination registers of Comblock use a clock frequency that exceeds the path delay of the SFU accelerator. Since the values on the registers are going to remain unchanged for more than one clock cycle, due to the software/hardware latency, it gives enough time to capture the right values. In this way, the proper functionality of the hardware is maintained. The results, which are expected, also show that the clock (`pl_clk0`) has a higher fanout of 8,851 compared to the reset and enable of the saboteur controller, (`FI_EN`, `FI_RST`), with 1,963 for both. Fault injection performance comparison for Trigonometric special functions core can be seen in Table 5.21. The principle of bit-banging method, using GPIO controls for the scan chain configuration versus the saboteur controller usage for scan chain configuration, are displayed. The overall configuration time for the bit-banging

Operation	Bit-Bang (GPIO)	Hardware Controller	Speedup
Fault Injection Campaign (including circuit execution):			
Single Fault Injection	2.692 s	1.390 s	1.94×
- Scan Configuration	1.310 s	0.008 s	164×
- Circuit Execution	1.382 s	1.382 s	1.00×
Baseline Performance:			
Golden Run (No Faults)	1.382 s	1.382 s	1.00×
Configuration Details:			
Scan Chain Length	1,963 bits		
Configuration Method	Python GPIO	Wishbone + 100 MHz shift	-
Pure Hardware Shift Time	-	19.63 µs (measured)	-

Table 5.21: Fault Injection Performance Comparison for SFU-Trigonometric: Bit-Banging vs Hardware Controller

method is 2.692 s. Using the hardware saboteur controller for configuration result in a speedup of 1.94 times, with a total time of 1.390 second. The total shift time is 0.008 s, with data polling during the wishbone interface interactions being the overhead. This speedup in configuration time shows the efficiency of the saboteur controller. The timing path characteristics of the implementation can be seen in

Path Component	Delay (ns)	Description
Total Path Delay	48.243	100%
Logic Delay	23.307	48.3%
Net Delay	24.936	51.7%
Clock Skew	0.125	-
Clock Uncertainty	0.112	-
Setup Slack	-43.176	Critical violation
Logic Levels	173	
High Fanout Net	2,301	`FI_TFen`

Table 5.22: Timing Path Characteristics of SFU1 Implementation.

Table 5.22. This report shows that the worst path shows a total delay of 48.243 ns. Both net delay at 51.7% and logic delay at 48.3% being a contributor to this delay. Due to the software/hardware latency, the implementation is treated as successful

as software control gives enough time for propagation before beginning of the next operation. The summary of FPGA primitive utilization of special functions

Primitive Type	**Count**	**Function**
LUT6	6,422	6-input lookup tables
FDRE	5,819	Edge-triggered flip-flops with reset
LUT5	2,699	5-input lookup tables
FDCE	2,219	Edge-triggered flip-flops with CE
LUT3	1,984	3-input lookup tables
LUT4	1,687	4-input lookup tables
LUT2	1,404	2-input lookup tables
RAMD32	742	Distributed RAM
FDSE	322	Flip-flops with set
LUT1	310	1-input lookup tables
MUXF7	239	7-input multiplexers
SRL16E	192	16-bit shift registers
SRLC32E	183	32-bit shift registers
CARRY8	74	Fast carry logic
RAMB36E2	60	Block RAM
DSP48E2	9	DSP slices

Table 5.23: FPGA Primitive Utilization Summary of SFU1 Implementation.

unit core implementation can be seen in Table 5.23. The balanced amount of edge-triggered flip flops with reset at 5,819 and 6-input look-up tables at 6,422 shows that the design utilizes both combinational logic and registered outputs. Low usage of DSP slices at 9 shows that the implementation relies on look-up tables for its arithmetic calculations. 239 MUXF7 multiplexers show that the implementation utilizes multiple data selection structures. The main usage of 60 RAMB36E2 block RAM instances are utilized on the ComBlock interface. Design complexity and verification coverage results at Table 5.24 show that the total logic nets utilized are 40,298, with 21,927 routable nets were successfully routed with zero errors. CLB utilization at 29.47% shows a resource efficient implementation. Functional correctness can be seen from zero unconstrained endpoints.

Metric	Value	Description
Total Logic Nets	40,298	
Routable Nets	21,927	
Fully Routed Nets	21,927	
Nets with Errors	0	
Unique Control Sets	520	
CLB Utilization	29.47%	2,599 of 8,820
Clock Regions Used	6	
Unconstrained Endpoints	0	
No Clock Pins	0	
Rent Exponent	0.36	

Table 5.24: Design Complexity and Verification Coverage of SFU1 Implementation.

5.5 Special Functions Unit 2

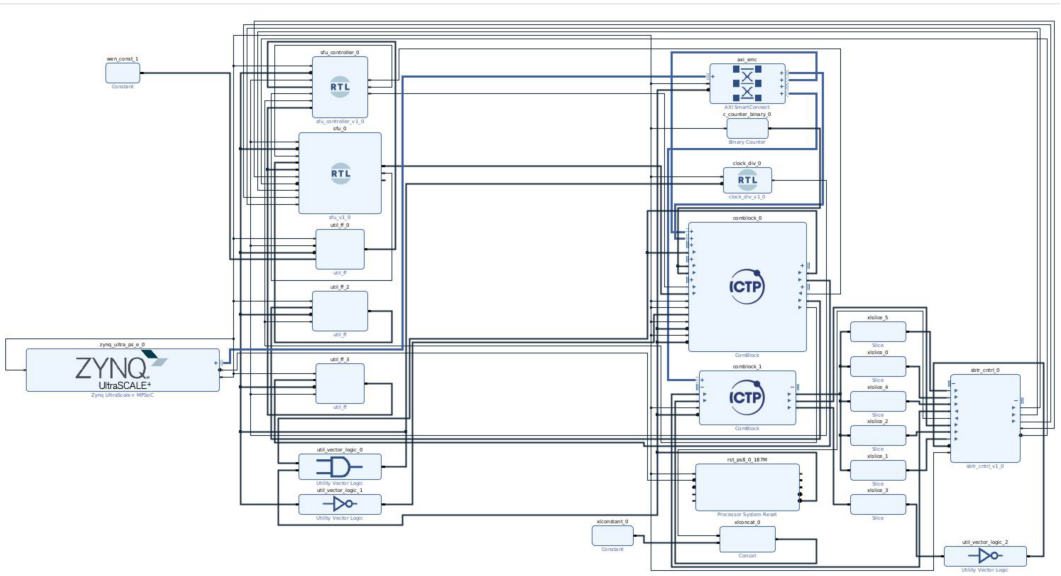


Figure 5.4: Special Function Unit Implementation.

The resource utilization report for the implementation of the special functions unit (CORDIC) core in Table 5.25 shows that the implementation uses only 16.52% of the CLB LUTs and 5.86% of CLB Registers available for its implementation. The usage

Resource Type	Used	Available	Util. (%)	Function
CLB LUTs	11,657	70,560	16.52	Logic implementation
- LUT as Logic	10,856	70,560	15.39	Combinational logic
- LUT as Memory	801	28,800	2.78	Distributed RAM/SRL
CLB Registers	8,271	141,120	5.86	Sequential elements
- Register as Flip Flop	8,271	141,120	5.86	State storage
- Register as Latch	0	141,120	0.00	Not used
Block RAM Tiles	60	216	27.78	Memory blocks
- RAMB36E2	60	216	27.78	Data buffers
- RAMB18	0	432	0.00	Not utilized
DSP Slices	16	360	4.44	Arithmetic operations
CARRY8	72	8,820	0.82	Fast carry chains
F7 Muxes	27	35,280	0.08	Wide multiplexers

Table 5.25: FPGA Resource Utilization for SFU2 (CORDIC Core) implementation.

of only 4.44% of DSP slices and 0.82% of available CARRY8 elements shows that the accelerator uses minimal dedicated hardware blocks for its arithmetic logic. The Block RAM tile usage is at 27.78% which is dedicated to the ComBlock interface and it is consistent with other benchmarks. The module-level resource allocation report

Module	LUTs	Registers	BRAM	DSP
`sfu_0`	5,737	1,855	0	16
- `cordic_inst`	1,607	105	0	4
- `exp2_inst`	1,138	774	0	2
- `log2_inst`	1,523	975	0	2
- `rsqrt_inst`	1,426	0	0	8
`axi_smc`	4,974	5,525	0	0
`comblock_0`	615	428	60	0
`sbtr_cntrl_0`	199	147	0	0
`comblock_1`	75	146	0	0
`sfu_controller_0`	5	4	0	0

Table 5.26: Detailed Module-Level Resource Allocation of SFU2 implementation.

in Table 5.26 shows the total resource allocated to each module. The main core

logic `sfu_0` is distributed across many modules named `cordic_inst`, `exp2_inst`, `log2_inst` and `rsqrt_inst` is the largest contributor to resource allocation with 5,737 look-up tables and 16 DSP slices. Inside the infrastructure, the `axi_smc` is allocated 4,974 look-up tables, while the (`sbtr_cntrl_0`) remains highly efficient, consuming only 199 look-up tables and 147 registers. This shows once again that the controller adds minimal overhead while being extremely efficient. The on-chip

Component	**Dynamic (W)**	**Static (W)**	**Total (W)**	**Percentage**
PS8 Processor System	2.807	0.104	2.911	83.10%
CLB Logic	0.101	0.061	0.162	4.62%
Signals	0.073	N/A	0.073	2.08%
Block RAM	0.043	0.001	0.044	1.26%
Clocks	0.046	0.000	0.046	1.31%
DSPs	0.004	0.000	0.004	0.11%
PL Static	-	0.221	0.221	6.31%
PS Static	-	0.104	0.104	2.97%
Total On-Chip	**3.179**	**0.324**	**3.503**	**100%**

Table 5.27: On-Chip Power Distribution Analysis of SFU2 implementation.

power distribution in Table 5.27 shows that the total power consumption is 3.503 W, with the PS8 processor system being the main contributor to the power consumption with 83.10% of total power. The CLB logic and the signals consume power at a rate of 4.62% and 2.08%, respectively. Throughout the benchmarks implementations, the processor side is responsible for the main power consumption and the bottleneck on the HyperFPGA fabric in terms of speed and power consumption. Table 5.28

Clock Resource	**Used**	**Fanout**	**Description**
`pl_clk0` (`BUFG_PS`)	1	8,656	Main system clock
`clk_out` (`BUFGCE`)	1	113	Derived clock from divider
`FI_EN` (`BUFGCE`)	1	1,749	Fault injection enable (non-clock loads)
`FI_RST` (`BUFGCE`)	1	1,747	Fault injection reset (non-clock loads)
Clock Frequency	187.512 MHz (5.333 ns period)		
Setup Slack (WNS)	-15.429 ns		
Hold Slack (WHS)	0.00 ns		

Table 5.28: Clock Network Utilization and Timing of SFU2 implementation.

shows the summary of the fanout and the description of the clock resources. With clock frequency at 187.512 MHz, the design reports the same critical warning as the posit adder core, the Worst Negative Slack (WNS) of -15.429 ns. The reason for this is because the source and destination registers of Comblock use a clock frequency that exceeds the path delay of the SFU CORDIC accelerator. Since the values on the registers are going to remain unchanged for more than one clock cycle, due to the software/hardware latency, it gives enough time to capture the right values. In this way, the proper functionality of the hardware is maintained. The results, which are expected, also show that the clock (`pl_clk0`) has a higher fanout of 8,656 compared to the reset and enable of the saboteur controller, (`FI_EN`, `FI_RST`), at 1,749 and 1747 respectively. Fault injection performance comparison

Operation	**Bit-Bang (GPIO)**	**Hardware Controller**	**Speedup**
Fault Injection Campaign (including circuit execution):			
Single Fault Injection	2.413 s	1.804 s	1.34×
- Scan Configuration	1.360 s	0.751 s	1.81×
- Circuit Execution	1.053 s	1.053 s	1.00×
Baseline Performance:			
Golden Run (No Faults)	1.053 s	1.053 s	1.00×
Scan Chain Length	1,749 bits		
Configuration Method	Python GPIO	Wishbone + 100 MHz shift -	

Table 5.29: Fault Injection Performance Comparison for SFU-CORDIC: Bit-Banging vs Hardware Controller

for special functions CORDIC core can be seen in Table 5.29. The principle of bit-banging method, using GPIO controls for the scan chain configuration versus the saboteur controller usage for scan chain configuration, are displayed. The overall configuration time for the bit-banging method is 2.413 s. Using the hardware saboteur controller for configuration result in a speedup of 1.34 times, with a total time of 1.804 s. Data polling from the software wishbone interface interactions is the overhead. This speedup in configuration time shows a minor increase in the efficiency of the saboteur controller compared to other implementations. The timing path characteristics of the implementation can be seen in Table 5.30 explains the timing violation. This report shows that the path delay 20.408 ns results in -15.429 ns slack. The net delay dominates at 61.1% and with the logic delay at 38.9%, they contribute to this delay. Due to the software/hardware latency, the implementation is treated as successful as software control gives enough time for propagation before beginning of the next operation. The summary of FPGA

Path Component	Delay (ns)	Percentage
Total Path Delay	20.408	100%
Logic Delay	7.937	38.9%
Net Delay	12.471	61.1%
Clock Skew	0.053	-
Clock Uncertainty	0.112	-
Setup Slack	-15.429 ns	Timing Violation
Logic Levels	106	
High Fanout Net	1,757	`FI_TFen`

Table 5.30: Timing Path Characteristics (Worst Path) of SFU2 implementation.

Primitive Type	Count	Function
FDRE	5,834	Edge-triggered flip-flops with reset
LUT6	5,038	6-input lookup tables
LUT5	2,614	5-input lookup tables
FDCE	2,113	Edge-triggered flip-flops with CE
LUT3	2,069	3-input lookup tables
LUT4	1,731	4-input lookup tables
LUT2	1,101	2-input lookup tables
RAMD32	742	Distributed RAM
FDSE	322	Flip-flops with set
LUT1	316	1-input lookup tables
SRL16E	194	16-bit shift registers
SRLC32E	183	32-bit shift registers
RAMB36E2	60	36Kb Block RAM
DSP48E2	16	DSP slices

Table 5.31: FPGA Primitive Utilization Summary of SFU2 implementation.

primitive utilization of special functions unit (CORDIC) core implementation can be seen in Table 5.31. The balanced count of edge-triggered flip flops with reset at 5,834 and 6-input look-up tables at 5,038 shows that the design utilizes both combinational logic and registered outputs. The utilization 377 shift register with high number of bits and the 16 DSP slices tracks with characteristics of a CORDIC core operation. The main usage of 60 RAMB36E2 block RAM instances are utilized

on the ComBlock interface. Design complexity and verification coverage results at

Metric	**Value**	**Description**
Total Logic Nets	43,906	
Routable Nets	20,551	
Fully Routed Nets	20,551	
Nets with Errors	0	
Unique Control Sets	525	
CLB Utilization	25.12%	2,216 of 8,820
Clock Regions Used	6 of 6	
Unconstrained Endpoints	316	
No Clock Pins	106	

Table 5.32: Design Complexity and Verification Coverage of SFU2 implementation.

Table 5.32 show that the total logic nets utilized are 43,906, with 20,551 routable nets were successfully routed with zero errors, indicating a clean implementation. CLB utilization at 25.12% shows a resource efficient implementation. The 316 unconstrained endpoints and 106 register pins driven by non-clock logic shows that there are areas for improvement in the implementation of the CORDIC core.

Chapter 6

Conclusion

In this work, the methodology of accelerating the reliability assessment of hardware accelerators through emulation using hyperscale systems is presented. The background research on the thesis started with the fundamental concepts of what will be used in order to analyze in this thesis. The general idea behind faults, fault modeling, and the different models to be used are explained. The scan chain configuration, how the faults are shifted, what are the main hardware pieces to be used are explained. Continuing with the emulation environment, HyperFPGA, is explained with detailing the pieces inside. Lastly, the hardware enabling the connection between the software and the hardware, the Core ComBlock, is explained.

The main task this thesis achieved is the development of a hardware saboteur controller that is implemented on Verilog in order to achieve significant speed-up during fault injection campaigns. The controller is developed utilizing the Wishbone B4 BUS protocol. A simple master-slave interface to achieve significant connection through acknowledgment. This way the fault injection can be achieved with single cycle precision. A finite state machine with nine states handles all the interaction with the software that is using ComBlock and the fault injection through the saboteur outputs that controls scan chain. This hardware can be implemented using minimal resources, at maximum 200 LUTs and 150 registers. This results in less than 2.7 percent overhead on the general digital circuit design. The design provides models of stuck-at-1, stuck-at-0 and bit-flip configurable for multiple chain lengths.

The design is tested first on a basic posit adder core, for creating a basis on implementation using ComBlock and emulation in hyperscale systems. After that, 4 additional circuits, the Tensor Core Unit, Stereo Vision Core and 2 Specialized Function Units are tested and the results published. The results of the hardware emulation can be summarized as:

- Performance Characteristics The evaluation of the emulation time proves that

the software control unit is the primary bottleneck of the design. During the operation, the saboteur controller achieved significant speed-up however the Python machine code, especially the data management algorithm, consume 99.87 percent of total execution time. The achieved speed-up varied greatly for different benchmark hardware implementations. The highest speed-up is shown to be the Tensor Core Unit with 7.05 times faster and Stereo Vision Core at 3.47 times, while the Special Function Units showed lower, 1.94 and 1.34 respectively. The main reason for this is the difference in execution times per the test vectors and the total number of test vectors at hand.

- Resource Utilization Patterns

  The results show that the hardware is greatly efficient for usage in fault injection campaigns.Total resource usage including saboteurs ranged from 16.52 percent to 57.64 percent of available LUTs. This usage stems from the complexity of the accelerator at hand and the controller for the accelerator, not the saboteur controller, which routinely showed less than 2.7 percent overhead in terms of resource usage.

  Block RAM usage is attributed to 27.78 percent with 60 RAMB36E2 blocks. This usage is directly connected to the RAM need inside the two ComBlock interface that is utilized. Depending on the size of the accelerator, memory usage can be the main bottleneck of the design.

- Power Analysis

  The processor system is the main dissipator of power, with ranging values from 63 to 83 percent, while the programmable logic with the fault injection consumes between 200 and 1,303 mW of dynamic power. The low power dissipation enables extended fault campaigns without any thermal concern or power leakage.

6.1 Future Work

After analyzing the results and the shortcomings of the work, in order have a more developed fault injection campaign these acknowledgments can be made:

- Software optimization: The results show that the bottleneck is the software input and output evaluation in terms of speed. A more comprehensive and optimized software campaign can be developed in Python or C/C++. If developed together with hardware improvements, multiple parallel fault injection campaigns can be made with an improved controller for multiple accelerators.

- Hardware improvement: In order to have an improvement on hardware design, multiple output ports for multiple scan-chain lengths can be developed in order to have parallel processing for different accelerators.

- Machine Learning Integration: Incorporating ML techniques to predict critical fault locations can be utilized for reducing the number of faults to be injected, decreasing the total campaign time.

Bibliography

[1] Carson Dunbar and Kundan Nepal. *Single stuck-at fault at an internal node of a circuit*. Online Image from "Using Platform FPGAs for Fault Emulation and Test-set Generation to Detect Stuck-at Faults". Accessed: 2025-10-01. 2011. URL: https://www.researchgate.net/figure/Single-stuck-at-fault-at-an-internal-node-of-a-circuit_fig1_220405407 (cit. on p. 13).

[2] Tiago Santos, Fernanda Lima Kastensmidt, and Luciano Ost. *SEU bit-flip simulation mechanism*. Online Image from "RASoC: A fault-tolerant router for networks-on-chip". Accessed: 2025-10-01. 2011. URL: https://www.researchgate.net/figure/SEU-bit-flip-simulation-mechanism_fig2_220850634 (cit. on p. 14).

[3] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. «Fault Injection into VHDL Models: The MEFISTO Tool». In: *Proc. IEEE 24th International Symposium on Fault-Tolerant Computing*. Austin, TX, June 1994, pp. 66–75 (cit. on p. 15).

[4] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante. «FPGA-Based Fault Injection for Microprocessor Systems». In: *Proc. 7th IEEE International On-Line Testing Workshop*. Taormina, Italy, July 2001, pp. 172–174 (cit. on p. 15).

[5] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, S. Pastore, and G. R. Sechi. «Evaluation of Single Event Upset Mitigation Schemes for SRAM Based FPGAs Using the FLIPPER Fault Injection Platform». In: *Proc. 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*. Rome, Italy, Sept. 2007, pp. 105–113 (cit. on p. 15).

[6] Sensoz Oguz. «Design, reliability evaluation, and hardening of visionoriented hardware accelerators». MA thesis. Torino: Politecnico Di Torino, 2025 (cit. on p. 15).

[7] ICTP-MLAB. *HyperFPGA-BSP*. GitLab repository. Accessed: 2025-10-01. 2024. URL: https://gitlab.com/ictp-mlab/hyperfpga-bsp (cit. on pp. 16, 17).

[8] Trenz Electronic GmbH. *TE0803 Technical Reference Manual*. Version 30. Trenz Electronic GmbH. 2021. URL: https://www.trenz-electronic.de/trenzdownloads/Trenz_Electronic/Modules_and_Module_Carriers/5.2x7.6/TE0803/REV03/Documents/TRM-TE0803-03.pdf (cit. on p. 16).

[9] Rodrigo Melo. *Core-ComBlock*. GitLab repository. Accessed: 2025-10-01. 2018. URL: https://gitlab.com/rodrigomelo9/core-comblock (cit. on pp. 18–20).

[10] Richard Herveille. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Specification Rev. B.4. Accessed: 2025-10-03. OpenCores, Sept. 2010. URL: https://cdn.opencores.org/downloads/wbspec_b4.pdf (cit. on p. 23).