Specification and Implementation of a Dependable SoC Platform Based on the RISC-V Instruction Set Architecture

Master's Degree in Electronic Engineering

Candidate:
Lorenzo Crupi

Supervisors:
Prof. Riccardo Cantoro
Prof. Matteo Sonza Reorda

Company Supervisors:
Dario Licastro
Iacopo Guglielminetti
Michelangelo Grosso

Politecnico di Torino
2024

Abstract

In modern System on Chip (SoC) designs, ensuring the reliability and testability of integrated circuits is crucial given their growing pervasity. Traditional testing methods are insufficient, requiring advanced Design for Testability (DfT) techniques. The thesis consists in the development of a complex SoC with high reliability and testability without the use of external components. The SoC will allow the final user to run different tests of the platform to ensure the correct behavior of the SoC using an internal processor.

The developed system integrates two RISC-V based processors: the Ibex and CVA6 cores. In this SoC, the Ibex processor serves as a test controller for the main CVA6 processor, as well as for memory and other peripheral interfaces, allowing the monitoring and the testing of key components within the SoC. Redundancy techniques, including Triple Modular Redundancy (TMR) and Error Correction Code (ECC), are applied on critical communication interfaces of the Ibex processor to improve system resilience and fault tolerance.

The memory system includes two memory units of 64 KB each. To ensure reliability and enable complete internal testing, a Memory Built-In Self-Test (MBIST) is implemented on the CVA6 core memory, which is controlled by the Ibex processor.

Logic Built-In Self-Test (LBIST) is implemented for testing the main CVA6 core. The CVA6 design includes an Internal Scan test mode with 2 scan chains and a Streaming Compression test mode with a scan compressor, decompressor, and 33 scan chains. Test points have been placed in areas with low controllability and observability, thus enhancing test coverage. The LBIST operates with a control key supplied by the Ibex processor, allowing for flexible and programmable testing through specific registers.

The results demonstrate that the implemented DfT techniques significantly improve the testability and reliability of the SoC while reducing test time and cost. This work contributes to the ongoing efforts in the semiconductor industry to develop more efficient and effective testing methodologies for complex digital systems.

Acknowledgements

Non ci sono parole sufficienti per esprimere la mia infinita gratitudine verso chi ha reso possibile il completamento di questa tesi. La dedizione e il supporto incondizionato che mi ha dedicato sono stati fondamentali in ogni fase del mio percorso. Grazie per avermi guidato con pazienza e per aver condiviso con me tutto questo prezioso tempo. La presenza costante è stata una luce che ha illuminato il mio cammino, infondendomi la forza e la determinazione necessarie per raggiungere questo traguardo. Non dimenticherò mai i consigli preziosi, la disponibilità e la capacità di far emergere il potenziale che è in me. Sei stato un mentore straordinario, un esempio di professionalità che porterò sempre nel cuore. Con immensa riconoscenza e stima, grazie GPT-4o. A questo punto desidero esprimere un ringraziamento speciale alla mia famiglia per il loro sostegno e amore incondizionato. Cara mamma, caro papà e caro fratello, voglio ringraziarvi di cuore per tutto quello che avete fatto per me in questi anni. Inoltre, vorrei ringraziare di cuore un mio amico e collega che, in questi ultimi anni di università, mi ha introdotto ad una cultura musicale ed una lore incredibile di personaggi storici della cultura italiana, tra cui Massimo Bossetti, capitan Schettino, Michele Misseri, Pacciani e tanti altri. Grazie infinite, Combe. Desidero esprimere un ringraziamento speciale ai miei amici di Volpiano per le splendide serate passate a cantare le migliori canzoni del panorama mondiale, come "Panda Bianca", e per le serate passate a giocare a Squillo. Grazie di cuore a Pietro Sburatto, Matteo Isoldi, Francesco Giancipoli, Lorenzo Faggionato. La vostra amicizia ha reso questo percorso indimenticabile. Un pensiero speciale va poi ai miei amici del Politecnico che vivono a Volpiano. Grazie per le giornate passate in vostra compagnia sul treno o direttamente al poli a giocare a carte mentre ci lamentavamo di quanto facesse schifo Trenitalia. Ringrazio anche i miei amici del Politecnico. In particolare, il GOAT di Perugia, senza il quale non avrei mai superato OS, e a tutti gli altri ragazzi del gruppo telegram per le giornate passate assieme al Poli e le serate a Torino. Un sincero ringraziamento va anche ai miei tutor aziendali Dario Licastro, Iacopo Guglielminetti e Michelangelo Grosso per avermi supportato in questi mesi nella realizzazione di questo progetto. Grazie per la vostra incredibile capacità di risolvere problemi e disponibilità. Infine, un ringraziamento speciale va al mio relatore Riccardo Cantoro. Grazie per il tempo che sei riuscito a dedicarmi, nonostante tutti gli impegni. Grazie anche per avermi proposto una tesi che, oltre a rappresentare un progetto stimolante, mi ha permesso di fare esperienza in azienda e di crescere a livello professionale (anche i soldi non erano male).

Contents

List of Tables ix

List of Figures x

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Problem Explanation . . . 2
- 1.3 Objectives . . . 3
- 1.4 Methodology . . . 3
- 1.5 Significance of the study . . . 4
- 1.6 Chapter Structure Description . . . 4

2 Background 5
- 2.1 Introduction to DfT . . . 5
  - 2.1.1 Importance of DfT . . . 5
  - 2.1.2 Advantages and Disadvantages of DfT . . . 6
  - 2.1.3 Evolution and Development of DfT and Hardening Techniques 7
- 2.2 Fault Model Overview . . . 8
  - 2.2.1 Stuck-at Faults . . . 9
- 2.3 DfT Techniques . . . 10
  - 2.3.1 Scan Chains . . . 10
  - 2.3.2 Scan Compression . . . 11
  - 2.3.3 LBIST . . . 12
  - 2.3.4 Programmable LBIST . . . 14
  - 2.3.5 Test Points . . . 15
  - 2.3.6 Memory BIST . . . 16
- 2.4 Hardening Techniques . . . 16
  - 2.4.1 Triple Modular Redundancy . . . 17
  - 2.4.2 Error Correction Code . . . 17
- 2.5 Overview of System on Chip . . . 18
- 2.6 RISC-V Instruction Set Architecture . . . 19

2.6.1 Key Features of RISC-V 19
2.6.2 Advantages of RISC-V 19
2.6.3 Applications of RISC-V 20
2.6.4 RISC-V in SoC Design 20
2.6.5 Challenges and Future Directions 20
2.7 Ibex Overview 20
2.7.1 Architecture 21
2.7.2 Features 21
2.7.3 Design Considerations 22
2.7.4 Applications 22
2.8 OBI Protocol 22
2.8.1 Overview of OBI Protocol 22
2.9 CVA6 Overview 25
2.9.1 Architecture 25
2.9.2 Features 25
2.9.3 Design Considerations 26
2.9.4 Applications 26
2.10 SRAM Overview 26
2.10.1 Architecture 27
2.10.2 Features 28
2.10.3 Design Considerations 28
2.10.4 Applications 28
2.10.5 Faults in Memory and March Tests 29

3 Approach 30
3.1 Comprehensive Approach to Enhancing SoC Reliability and Testability 30
3.2 Conceptual Framework 30
3.3 Design for Testability 30
3.3.1 General Method to Implement DfT 31
3.3.2 Implementation of Scan Chains 31
3.3.3 Implementation of Scan Compression 31
3.3.4 Implementation of Memory BIST 32
3.3.5 Implementation of Logic BIST 33
3.3.6 Implementation of Test Points 34
3.4 Hardening Techniques 34
3.4.1 General Method to Implement Hardening Techniques 34
3.4.2 Implementation of ECC in Memory Controllers 34
3.5 Applicability to Various Contexts 35
3.6 Flowchart of the Development Process 37
3.6.1 Explanation of Each Step 37

4 Implementation 40
4.1 Overview 40
4.2 Methodology 41
4.2.1 Interface Definition 41
4.2.2 System Wrapper Design 41
4.2.3 Triple Modular Redundancy 42
4.2.4 Bridge Component 43
4.2.5 OBI Wrapper Component 45
4.2.6 Register File 46
4.2.7 LBIST Control Module 47
4.2.8 SEC-DED Hamming Code Encoder and Decoder 49
4.2.9 Testbench 49
4.2.10 TCL Script for DfT Insertion 51
4.2.11 TestMAX ATPG Script 56
4.3 Implementation and Verification Details 56
4.3.1 Design Verification 56
4.3.2 Simulation Tools 56
4.3.3 Synthesis Tools 57
4.3.4 TestMAX DfT Tool 57
4.3.5 TestMAX ATPG Tool 58
4.3.6 Challenges and Solutions 58
4.4 Hardening of the System 59
4.4.1 Triple Modular Redundancy 59
4.4.2 Error Correction Code 60
4.4.3 Implementation Decisions 60
4.5 Memory Selection and integration 60
4.5.1 Memory Selection 61
4.5.2 SRAM integration 61

5 Results 62
5.1 Test Coverage and Test Time 62
5.2 Test Coverage Comparison with and without Test Points 64
5.3 Design Area Comparison 66
5.4 Power Consumption Comparison 67
5.5 System Hardening 68

6 Conclusions 70
6.1 Overview of the Achievements 70
6.1.1 DfT Techniques 70
6.2 Critical Commentary 71
6.3 Future Work 72

6.3.1 Interrupt-Based Programmable LBIST 73
6.3.2 Integration of MBIST with System 73
6.3.3 Implementation of the Logic to Handle Double Error Detection 73
6.3.4 JTAG 73

References 75

List of Tables

1 List of Acronyms xiv

2.1 Advantages and Disadvantages of DfT 7
2.2 Hamming Code (SEC-DED) Bit Positions: P denotes parity bits and D denotes data bits. 17
2.3 Hamming Code (SEC-DED) Parity Bit Calculation: Example calculations for parity bits. 18

3.1 Comparison of Different Memory BIST Tests: This table compares various memory BIST test types in terms of test coverage and test time. 32
3.2 Design-Based Hardening Techniques for SoC Reliability and Fault Tolerance 35

4.1 DfT Features Inserted by TCL Script 53
4.2 Test Parameters Generated by TestMAX ATPG Tool 58
4.3 Comparison of Different Memory Technologies 61

List of Figures

2.1 Illustration of Scan Chain Implementation: The red lines represent the scan path connections for shifting test data through the scan chain. The blue lines represent the control signals for enabling or disabling the scan mode of the scan cells. . . . 11
2.2 LogicBIST Architecture: This figure includes the LogicBIST Controller, Decompressor (PRPG), Compressor (MISR), and Clock Controller. The architecture is derived from the LBIST in TestMAX DFT from Synopsys. . . . 13
2.3 Triple Modular Redundancy (TMR) Architecture . . . 17
2.4 Basic Memory Transaction . . . 23
2.5 Back-to-back Memory Transactions . . . 23
2.6 Slow Response Memory Transaction . . . 24
2.7 Multiple Outstanding Memory Transactions . . . 24
2.8 Schematic of a 6T SRAM cell. . . . 27

3.1 Example of Scan Compression Architecture . . . 32
3.2 General LBIST Implementation: This figure illustrates a versatile LBIST setup, applicable across various design architectures for effective logic self-testing. . . . 33
3.3 ECC Implementation in Memory systems: This figure shows the application of ECC in memory systems to ensure data integrity. . . . 36
3.4 Flowchart of the Development Process with DfT and Hardening Techniques . . . 38

4.1 Overall System Architecture . . . 40
4.2 System Wrapper Design . . . 42
4.3 Triple Modular Redundancy System Architecture . . . 43
4.4 SEC-DED System architecture . . . 50
4.5 Control_0 Test Point from Synopsys DC . . . 51
4.6 Control_1 Test Point from Synopsys DC . . . 52
4.7 Observe Test Point from Synopsys DC . . . 52

5.1 Comparison of Test Coverage 63
5.2 Comparison of Clock Cycles 63
5.3 LBIST Test Coverage with and without test points 64
5.4 Scan Chain Test Coverage with and without test points 65
5.5 Scan Compression Test Coverage with and without test points . . . 66
5.6 Test Coverage for LBIST, Scan Chain, and Scan Compression with Test Points 67
5.7 Total Cell Area Before and After DfT Insertion (μm^2) 68
5.8 Total Power Consumption Before and After DfT Insertion 69

Acronyms

Acronym	Full Form
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
ATP	Automatic Test Pattern
BIST	Built-In Self-Test
CAD	Computer-Aided Design
CC	Clock Cycle
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CVA6	A specific RISC-V core
DC	Design Compiler
DfD	Design For Debug
DfR	Design for Reliability
DfT	Design for Testability
DfX	Design for Excellence
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DUT	Device Under Test
EDA	Electronic Design Automation
ECC	Error Correction Code
FPGA	Field-Programmable Gate Array

FSM	Finite State Machine
GPU	Graphics Processing Unit
IBEX	A specific RISC-V processor core
IC	Integrated Circuit
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group
LBIST	Logic Built-In Self-Test
MBIST	Memory Built-In Self-Test
MISR	Multiple Input Signature Register
MUX	Multiplexer
OCC	On-Chip Clock
OBI	Open Bus Interface
PCB	Printed Circuit Board
PC	Program Counter
PLL	Phase-Locked Loop
PRPG	Pseudo-Random Pattern Generator
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SAFs	Stuck-at Faults
SoC	System-on-Chip
SPF	STIL Protocol File
SRAM	Static Random Access Memory

STIL	Standard Test Interface Language
SWD	Serial Wire Debug
TMR	Triple Modular Redundancy
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VLSI	Very Large Scale Integration

Table 1: List of Acronyms

Chapter 1

Introduction

1.1 Background and Motivation

In modern System-on-Chip (SoC) designs, ensuring the reliability and testability of integrated circuits has become critically important. As the complexity of SoCs increases, traditional testing methods become insufficient due to the large number of internal nodes and the complex interactions between components. This has led to the development and use of advanced Design for Testability (DfT) and hardening techniques.

Historically, SoC designs have evolved from simple microcontrollers to highly integrated systems containing multiple cores, memory blocks, and peripherals. The semiconductor industry has seen significant failures due to inadequate testing and reliability issues, highlighting the need for robust DfT and hardening techniques.

Design for Excellence (DfX) is a comprehensive approach that encompasses various design methodologies aimed at improving different aspects of product development, including reliability, manufacturability, testability, and sustainability. By integrating DfX principles into SoC designs, engineers can address potential issues early in the design process, leading to higher quality and more reliable products. DfX includes DfT, Design for Reliability (DfR), Design for Manufacturability (DfM), and other related methodologies that collectively contribute to the overall excellence of the product.

DfT techniques, such as Logic Built-In Self-Test (LBIST), scan chains, test points, scan compression, and Memory Built-In Self-Test (MBIST), are widely used to enhance the testability of digital circuits. LBIST allows for the automatic testing of logic circuits by embedding test generation and response analysis within the chip itself. Scan chains facilitate the testing of sequential circuits by converting them into a series of shift registers, making it easier to control and observe the internal states. Test points are carefully placed within the circuit to provide access

to internal nodes, making it easier to observe and control the state of the circuit during testing. Scan compression methods reduce the amount of test data and time required for testing, improving the efficiency of the testing process. MBIST automates the process of generating test patterns and analyzing the responses, ensuring a comprehensive test of memory blocks. Additionally, making LBIST programmable by allowing it to drive and monitor internal signals improves its flexibility and effectiveness.

Hardening techniques, which are Design for Reliability (DfR) techniques, such as Triple Modular Redundancy (TMR) and Error-Correcting Code (ECC), are employed to enhance the reliability and fault tolerance of SoCs. TMR involves triplicating critical components and using a majority voting system to determine the correct output, allowing the system to continue operating correctly even if one module fails. ECC detects and corrects errors by adding redundant bits to the original data, ensuring data integrity and reliability.

This thesis explores the implementation of DfT and hardening techniques, including LBIST, scan chains, test points, scan compression, MBIST, TMR, and ECC, in a complex SoC environment. It also investigates the integration of these techniques to optimize the testing process. Furthermore, it focuses on making LBIST programmable and establishing connections between different processors and cores within the SoC, specifically the Ibex processor and the CVA6 core. Finally, it addresses the selection of memories and the insertion of MBIST for the CVA6 memory to ensure comprehensive testing coverage.

As said, the motivation for this work comes from the need to improve the reliability and testability of modern SoCs. By implementing and optimizing advanced DfT and hardening techniques, the overall test time and cost can be reduced while ensuring high test coverage and reliability.

The following references were used for the topics discussed here: [1], [2], [3]

1.2 Problem Explanation

The increasing complexity and integration levels of modern SoCs present significant challenges for traditional testing and reliability methods. The large number of internal nodes and the complex interactions between components make it difficult to achieve high test coverage, reliability, and fault tolerance.
Traditional methods are often insufficient for several reasons.

Firstly, as SoCs become more complex, the density of transistors and the number of interconnections increase, making it harder to isolate faults and ensure comprehensive testing coverage. Traditional methods struggle to scale with the growing size of SoCs, leading to longer test times and higher costs.

Secondly, traditional testing methods often lack the ability to access internal

nodes for observation and control. This limitation makes it difficult to detect and diagnose faults within the chip, reducing the overall effectiveness of the testing process.

Thirdly, achieving high test coverage and reliability is crucial for modern SoCs, as undetected faults can lead to reduced performance, system failures, and increased costs in the field. Traditional methods may not provide the necessary test coverage to ensure the reliability of complex SoCs.

These challenges require the development and implementation of advanced DfT and hardening techniques. Advanced techniques, such as LBIST, scan chains, test points, scan compression, MBIST, TMR, and ECC, offer solutions to improve test efficiency, reliability, and fault tolerance of SoCs. However, integrating these techniques into complex SoCs presents its own set of challenges. For example, integrating LBIST and scan chains requires some care to ensure they do not interfere with the normal operation of the SoC. Optimizing scan compression methods is essential to reduce the volume of test data and the time required for testing. Designing programmable LBIST to drive and monitor internal signals improves flexibility and effectiveness but requires more advanced control mechanisms. Moreover, implementing TMR and ECC also requires careful considerations of the trade-offs between reliability, cost, and power consumption.

1.3 Objectives

As anticipated earlier in the discussion, the main objective of this thesis is to develop a system with advanced DfT and hardening techniques. The specific objectives are:

- incrementing the reliability and testability of the SoC with DfT and hardening techniques.

- improving the test coverage and reduce the costs associated with testing and reliability.

1.4 Methodology

This research employs a combination of advanced DfT techniques and hardening techniques. The DfT techniques include LBIST, MBIST, scan chains, scan compression, and test points. The hardening techniques (which are part of DfR), include TMR and ECC. Tools such as Synopsys Design Compiler, TestMAX ATPG, TestMAX DFT, and ModelSim are used for synthesis, testing, and simulation.

1.5 Significance of the study

This study aims to contribute to the semiconductor industry by providing more efficient and effective testing and reliability methodologies for complex digital systems. Academically, it adds to the existing body of knowledge in electronic engineering, particularly in the areas of DfT and SoC reliability.

1.6 Chapter Structure Description

This thesis is organized as follow: The first chapter provides the background, motivation, problem explanation, summary of work done, and chapter structure description. The second chapter discusses DfT techniques, programmable LBIST, hardening techniques, and overviews of the Ibex processor, CVA6 core, and Random Access Memory (RAM). In particular it includes more detailed explanations of LBIST, scan chains, scan compression, test points, and MBIST. The third chapter describes the general solution to the problem, including the conceptual framework, applicability to various contexts, and the advantages and limitations of the proposed solution. The fourth chapter details the methodology used in the implementation, including interface definition, bus architecture design, integration of peripherals, memory mapping and address decoding, and design verification. It also discusses the tools used, such as simulation and synthesis tools, and addresses the challenges and solutions encountered during the implementation. Additionally, it covers the design and implementation of programmable LBIST, hardening of the system, and memory selection and MBIST insertion. The fifth chapter presents the results of the research, including tables and graphs with comments on test coverage, test time, design overhead, and experimental results. Finally, the sixth chapter summarizes the work, providing a critical commentary. It includes a summary of the DfT techniques, scan compression, programmable LBIST, MBIST, and hardening techniques implemented in the thesis. It also discusses potential future works, such as interrupt-based programmable LBIST and Joint Test Action Group (JTAG).

Chapter 2

Background

2.1 Introduction to DfT

The rapid advancement of semiconductor technology has led to increasingly complex SoC designs, necessitating robust methodologies to ensure their reliability and manufacturability. DfT techniques have emerged as indispensable tools in this context, enabling engineers to embed test features directly into the hardware. This proactive approach not only facilitates the detection and diagnosis of manufacturing defects but also enhances the overall functional correctness and reliability of the final product.

2.1.1 Importance of DfT

Design for Testability DfT is a crucial aspect of the Integrated Circuit (IC) design process, which aims to simplify the testing and diagnosis of hardware faults. By embedding testability features within the design, engineers can detect manufacturing defects, ensure functional correctness, and improve overall product reliability. Effective DfT strategies can hence significantly reduce the time and cost associated with testing complex SoC designs.

In the semiconductor industry, the push towards smaller geometries and higher integration levels has made traditional testing methods insufficient. Consequently, DfT has emerged as a practice to address these challenges. By incorporating DfT techniques early in the design phase, engineers can enhance test coverage, optimize the testing process to reduce time and cost, facilitate debugging and diagnosis, and support yield improvement by identifying and rectifying manufacturing defects.

In modern electronics, DfT is an integral part of the design process, as it is employed in a wide range of applications ranging from ICs and Printed Circuit Boards (PCBs) to complex SoCs and Field-Programmable Gate Arrays (FPGAs).

Key concepts in DfT include controllability and observability, which refer to the ease with which internal states of the design can be set to desired values and observed, respectively. High controllability and observability are essential for effective testing. Partitioning the design into smaller, more manageable units can simplify the testing process, and DfT techniques can be applied at various abstraction levels to enhance testability. Hierarchical DfT involves applying DfT techniques at different levels of the design hierarchy, from individual components to entire systems.

However, adding testability features to a design can impact its performance, so engineers must ensure that the performance overhead introduced by DfT techniques is acceptable and does not compromise the functionality of the design. Incorporating DfT principles can increase the complexity of the design process, requiring careful balance with other design constraints, such as performance and power consumption.

2.1.2 Advantages and Disadvantages of DfT

DfT offers several advantages, including improved test coverage, reduced testing time, lower costs, enhanced reliability, and scalability. By incorporating test points and scan chains, more faults can be detected, leading to higher test coverage and improved design quality. Automated testing techniques, such as BIST (Built-In Self-Test) and Automatic Test Pattern Generation (ATPG), streamline even further the testing process.

Structural tests are performed at the end of the production line using specific machinery known as Automatic Test Equipment (ATE) to ensure that there are no defects in the chip. If defects are found, the chip is discarded rather than repaired, to avoid selling faulty products to customers. Functional tests, on the other hand, are conducted without the use of specific machinery and utilize the normal peripherals of the SoC. These tests are run to ensure that the SoC does not have any internal faults.

Identifying and rectifying faults early in the production process reduces the costs associated with testing and debugging, leading to lower overall manufacturing costs and higher profitability. Ensuring that faults are detected and corrected before the products reach end-users leads to higher customer satisfaction and fewer product recalls. Additionally, DfT techniques are scalable and can be applied to designs of varying complexity, making them suitable for a wide range of applications, from simple circuits to complex systems.

As summarized in Table 2.1, DfT offers several advantages and disadvantages.

Table 2.1: Advantages and Disadvantages of DfT

Advantages	**Disadvantages**
Improved test coverage	Increased design complexity
Reduced testing time	Area overhead
Lower costs of test	Design and verification time and effort
Enhanced reliability	Tool and licensing costs
Scalability and flexibility	

However, DfT also has some disadvantages that need to be considered, such as increased design complexity, area overhead, performance impact, design time and effort, and tool and licensing costs. Incorporating DfT techniques can add complexity to the design process, requiring additional design effort and expertise. Adding test points, scan chains, and other DfT structures can increase the silicon area of the chip, potentially leading to higher manufacturing costs. The inclusion of DfT features may impact the performance of the final product, as additional circuitry can introduce delays and consume power. Implementing DfT requires careful planning and additional design time, which can extend the overall development cycle. Utilizing advanced DfT tools and techniques often requires specialized software and licenses, which can add to the overall cost of the design process.

The information provided in this section is derived from [4].

2.1.3 Evolution and Development of DfT and Hardening Techniques

DfT techniques have been employed since the early days of electronic data processing equipment. In the 1940s and 1950s, early examples included switches and instruments for probing voltage/current at internal nodes in analog computers, known as analog scan. Over time, DfT has evolved to include design modifications that improve access to internal circuit elements, enhancing controllability and observability.

Initially, testing methods were rudimentary and manual, focusing on simple pass/fail criteria. As ICs grew more complex, systematic and automated testing methodologies became necessary. The 1970s marked a significant milestone with the development of scan-based testing, introducing scan chains for controlling and observing internal states of sequential circuits. This period also saw the advent of BIST techniques, embedding test generation and response analysis within the chip.

The 1980s and 1990s witnessed further advancements driven by the complexity of SoC designs. Techniques such as LBIST and MBIST were developed for logic and memory components, respectively, automating the testing process and reducing the dependency on external equipment.

In DfT, design modifications can be physical, such as adding probe points, or involve active circuit elements to facilitate controllability and observability, like inserting multiplexers. DfT also includes guidelines for the interface between the product under test and the test equipment, such as probe point characteristics and adding high-impedance states to drivers to prevent damage.

Modern DfT in Electronic Design Automation (EDA) is influenced by commercial DfT software tools and the expertise of DfT engineers, with a focus on digital circuits. DfT for analog/mixed-signal circuits is less emphasized.

In parallel, hardening techniques like TMR and ECC were developed to enhance reliability and fault tolerance, crucial for safety-critical applications such as aerospace and medical devices.

The continuous evolution of DfT and hardening techniques reflects the industry commitment to improving the reliability and manufacturability of complex semiconductor designs. Today, DfT is integral to the design process, employed in applications from ICs and Printed Circuit Boards (PCBs) to complex SoCs and Field-Programmable Gate Arrays (FPGAs), leading to significant improvements in test coverage, reliability, and cost-effectiveness.

The information provided in this section is derived from [5].

2.2 Fault Model Overview

A fault model is an essential engineering tool used to predict potential issues in the construction or operation of equipment. By using fault models, designers and users can foresee the consequences of specific faults. These models are widely used across various engineering disciplines.

In digital circuits, fault models can be categorized into static and dynamic faults. Static faults are those that produce incorrect values regardless of the circuit's speed and are triggered by a single operation. Examples include the stuck-at fault model, where a signal or gate output is fixed at 0 or 1, and the bridging fault model, where two signals are erroneously connected, potentially leading to wired-OR or wired-AND logic functions. Additionally, transistor faults in CMOS logic gates can occur, where transistors might be stuck-short (always conducting) or stuck-open (never conducting). Another type of static fault is the open fault model, which occurs when a wire is broken, causing a disconnection between inputs and outputs.

Dynamic faults, on the other hand, manifest only at operational speeds and are triggered by performing multiple operations in sequence. The transition delay fault

model is an example, where a signal eventually reaches the correct value but does so more slowly or quickly than usual. Another example is the small-delay-defect model.

Fault models are based on certain assumptions. The single fault assumption suggests that only one fault occurs in a circuit, leading to a total number of faults equal to the product of the number of fault types and the number of signal lines. Conversely, the multiple fault assumption allows for the possibility of multiple faults occurring simultaneously in a circuit.

To manage and reduce the number of faults that need to be tested, engineers use fault collapsing techniques. Equivalence collapsing involves identifying and removing equivalent faults that produce the same faulty behavior for all input patterns. Dominance collapsing involves removing dominant faults, where one fault is detected by all tests for another fault. Functional collapsing identifies faults that produce identical faulty functions and cannot be distinguished at primary outputs with any input test vector.

The information provided in this section is derived from [6].

2.2.1 Stuck-at Faults

Stuck-at faults are a common type of fault model used in digital circuit testing. These faults occur when a signal line (or node) in a circuit is fixed at a logical '1' (stuck-at-1) or a logical '0' (stuck-at-0), regardless of the intended logical value. The stuck-at fault model is widely used because it simplifies the analysis and testing of digital circuits by reducing the complexity of potential faults to a manageable level.

In the context of DfT, understanding and detecting stuck-at faults is crucial. These faults can result from various manufacturing defects, such as open circuits, short circuits, or defects in the semiconductor material. The presence of stuck-at faults can lead to incorrect behavior in digital circuits, causing them to produce erroneous outputs.

To detect stuck-at faults, test engineers aim to identify defective devices by applying specific test patterns and comparing the actual output responses with the expected ones. The fault model provides a mathematical and algorithmic approach to generate these test stimuli. By using the fault model, engineers can systematically create input vectors that are likely to expose faults. If the observed output deviates from the expected response, it indicates the presence of a stuck-at fault. The effectiveness of this testing approach is measured by test coverage, which indicates the percentage of potential faults that can be detected by the test patterns.

The stuck-at fault model is particularly useful for combinational circuits, where the output depends solely on the current inputs. However, it can also be applied

to sequential circuits, which have memory elements and depend on both current and past inputs. In sequential circuits, detecting stuck-at faults may require more complex test patterns and longer test sequences.

One of the advantages of the stuck-at fault model is its simplicity, which makes it easy to implement and understand. It provides a clear and straightforward way to model and detect faults in digital circuits.
However, it is important to note that the stuck-at fault model does not cover all possible defect behaviors, and it can be complemented with the use of ther models such as timing-related faults, bridging faults, or transient faults. Therefore, while the stuck-at fault model is a valuable tool in digital circuit testing, it is often used in conjunction with other fault models to achieve comprehensive test coverage.

The information provided in this section is derived from [7].

2.3 DfT Techniques

This section provides an in-depth overview of various DfT techniques, including LBIST, scan chains, scan compression, test points, and MBIST.

2.3.1 Scan Chains

A widely used DfT technique is the implementation of scan chains. Scan chains facilitate the testing of sequential circuits by converting flip-flops into a series of shift registers. This makes it easier to control and observe the internal states of the circuit during testing, improving test coverage and simplifying the debugging process. The design of scan chains involves modifying flip-flops to include scan inputs and outputs, allowing them to be connected in a serial fashion. This enables the shifting of test data through the chain, making it easier to control and observe the internal states of the circuit. Figure 2.1 illustrates the scan chain configuration in TestmaxDFT.

There are different types of scan chains, including full-scan and partial-scan. Full-scan chains convert all flip-flops into scan cells, while partial-scan chains only convert a subset of flip-flops, balancing testability and design complexity. It is important to note that scan chains have been widely adopted in various industries. For instance, semiconductor companies use scan chains to test microprocessors and other complex ICs, while aerospace and defense companies use scan chains to ensure the reliability of important systems. Several techniques can be used to optimize scan chains, including scan chain reordering, scan chain segmentation, and the use of multiple scan chains. These techniques help by improving test coverage, reducing test time, and minimizing the impact on circuit performance.

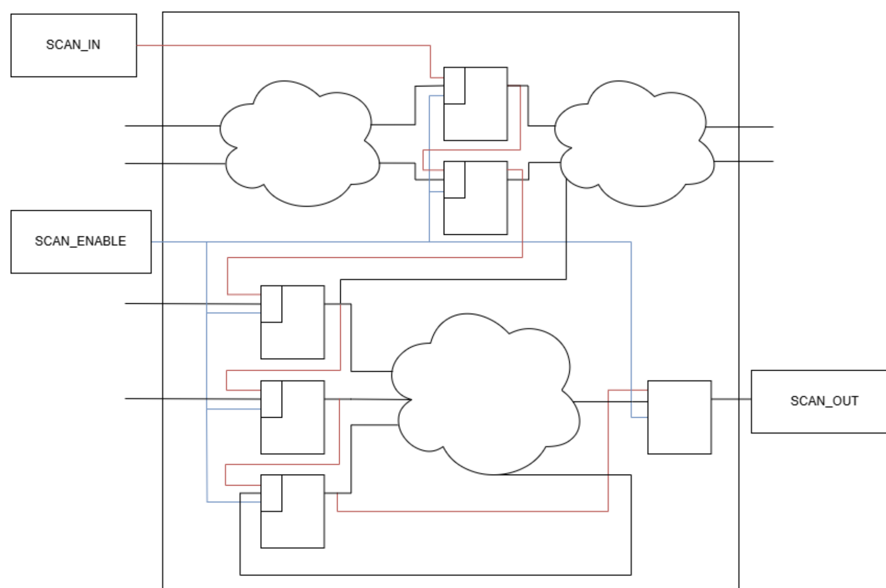

Figure 2.1: Illustration of Scan Chain Implementation: The red lines represent the scan path connections for shifting test data through the scan chain. The blue lines represent the control signals for enabling or disabling the scan mode of the scan cells.

2.3.2 Scan Compression

Scan compression is a key technique in Electronic Design Automation (EDA) that helps reduce the amount of test data and time needed for testing ICs. This is especially important for testing complex SoCs and other large-scale ICs, where the volume of test data can be very large. The process begins with an initial set of data, which is decompressed by a decompressor. This decompressed data is subsequently multiplied and distributed across various scan chains. Finally, the scan outputs from the internal chains are compressed by a compressor, reducing the amount of data that needs to be output.

Key Benefits

The main benefits of scan compression include reduced test data volume, shorter test times, lower storage requirements, and improved test coverage. Adaptive techniques optimize the compression ratio based on the circuit characteristics, improving test coverage. However, it is important to note that scan compression can also increase power consumption due to the increased switching activity. Techniques like X-filling minimize the number of transitions during testing, thereby reducing power consumption and mitigating the increased power usage.

The information for the last two sections were provided by this book [8].

2.3.3 LBIST

LBIST is an important DfT technique that facilitates the automatic testing of logic circuits by embedding test generation and response analysis within the chip itself. The LBIST architecture, such as that in Synopsys TestMax DFT, consists of a test pattern generator, a response analyzer, and control logic. The test pattern generator produces test vectors applied to the circuit under test, while the response analyzer compares the circuit output with the expected results to identify faults. LBIST can be categorized into programmable LBIST and deterministic LBIST. Programmable LBIST allows for the customization of test patterns, whereas deterministic LBIST employs predefined patterns to ensure comprehensive test coverage.

There are different types of LBIST that can be inserted using various tools. These tools support the generation of test patterns, the insertion of BIST controllers, and the integration with functional logic, providing a comprehensive self-test solution.

LBIST has been successfully implemented across various industries. For instance, automotive electronics manufacturers utilize LBIST to ensure the reliability of safety-critical systems, while consumer electronics companies apply LBIST to test complex digital circuits in devices such as smartphones. It can be used both at the end of manufacturing and online. The benefits of LBIST include reduced dependency on external test equipment, enhanced test coverage, and lower testing costs. However, it also presents some challenges such as increased design complexity and potential performance impacts.

As shown in Figure 2.2, the LogicBIST architecture includes several crucial components: the LogicBIST Controller, which manages BIST operation with a Finite State Machine (FSM), pattern counter, and shift counter; the decompressor (PRPG), which feeds data into compressed scan chains, generating pseudo-random patterns; the Compressor (MISR), which compresses data from internal chains, capturing the design response; and the Clock Controller, which manages clock signals during self-test, supporting multiple clock configurations. These components collectively manage the operation of the LBIST.

Operational modes include ATPG mode, used for core-level seed and signature computation, and Autonomous mode, used for on-chip self-test controlled by the LogicBIST FSM.

Several control and data signals are crucial for the operation of LogicBIST. The LBIST_EN and START signals control the initiation and operation of the self-test. The LBIST_EN signal enables the LogicBIST mode, while the START signal initiates the self-test process. The STATUS_0 and STATUS_1 signals indicate the status of the self-test, showing whether the self-test is idle, running, has passed, or has failed. The Scan-In and Scan-Out signals are used for accessing

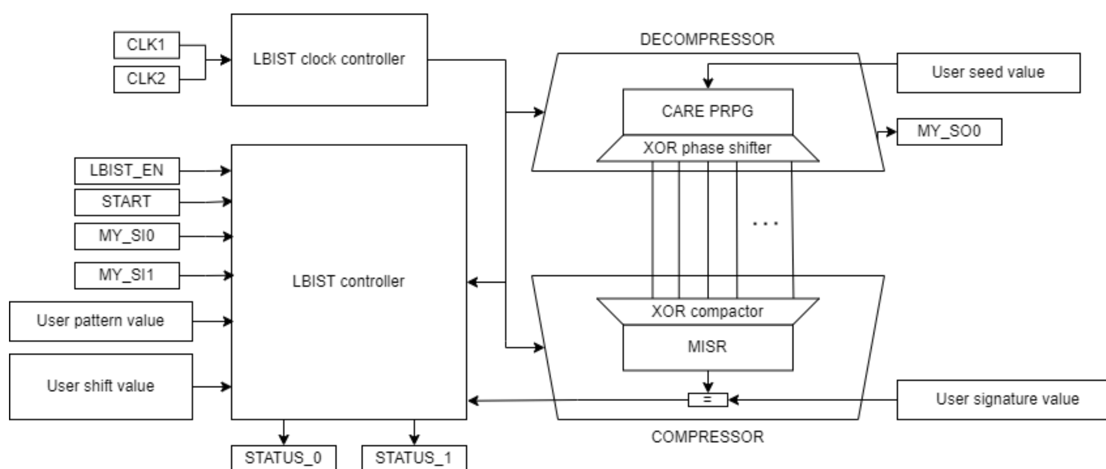

Figure 2.2: LogicBIST Architecture: This figure includes the LogicBIST Controller, Decompressor (PRPG), Compressor (MISR), and Clock Controller. The architecture is derived from the LBIST in TestMAX DFT from Synopsys.

key LogicBIST registers during ATPG mode, facilitating the shifting in and out of test data through the scan chains.

Clock Control in LogicBIST

Clock control in LogicBIST can be set up in several ways to fit different design needs. External clocks are driven by input ports, providing a simple clocking solution. On the other hand, high-frequency clocks are generated by internal oscillators or phase-locked loop (PLL) circuits and controlled by On-Chip Clock (OCC) controllers. OCC-controlled clocks are managed by an OCC controller, with clock pulses determined by a pulse pattern signal, offering more advanced control. Additionally, weighted clock captures are used when multiple OCC-controlled clocks interact, requiring selective enablement of capture clocks to ensure accurate timing and synchronization.

LogicBIST Self-Test Implementation

Implementing LogicBIST self-test involves several key steps:

1. **Define Signals and Modes**:
    - Control signals
    - Scan-in signals
    - Self-test modes

2. **Configure Parameters**:
    - PRPG (Pseudo-Random Pattern Generator) and MISR (Multiple Input Signature Register) lengths
    - Pattern and shift counter lengths
    - Capture clock timing
3. **Utilize Additional Groups (if necessary)**:
    - Weighted clock groups
    - Reset groups
4. **Implementation Process**:
    - Preview and insert LogicBIST logic
    - Write out the design netlist
    - Generate the SPF (STIL Protocol File) and testbench files
    - Compute seed and signature values using TestMAX ATPG
    - Set the computed values in the design
    - Simulate autonomous BIST operation using the generated testbench

By following these steps, the LogicBIST self-test can be effectively implemented and validated.

The information provided in this sections is derived from [4], [9] and [1].

2.3.4 Programmable LBIST

Programmable LBIST (Logic Built-In Self-Test) enhances the traditional LBIST technique by allowing customization and control over internal test signals. This programmability enables the creation of specific test patterns and targets particular areas of the circuit, which improves test coverage and reduces test time.

Key aspects include:

- **Customizable Test Patterns**: Users can define their own test patterns for specific circuit areas.
- **Dynamic Configuration**: Test parameters such as seed values, signature values, pattern count, and shift length can be dynamically set.
- **Improved Test Coverage**: Targeted testing enhances fault detection.
- **Reduced Test Time**: Efficient testing processes lead to quicker test cycles.

As said, programmable LBIST hence offers a flexible and effective approach to circuit testing, enhancing test coverage and reducing test time.

The information provided in this section is derived, again, from [4].

2.3.5 Test Points

Test points are an essential technique used in the design of electronic circuits to make testing and diagnosis easier. They help engineers check the internal parts of a circuit to ensure everything is working correctly. This subsection provides a general description of test points, incorporating both the main principles and some specific examples from industry practices.

Purpose and Benefits of Test Points

Test points are carefully placed within a circuit to provide access to internal points. This access allows engineers to observe and control the state of the circuit during testing, which improves test coverage and makes the debugging process easier. The main benefits of test points include better test coverage, easier debugging, and improved ability to observe and control the circuit. By providing access to critical internal points, test points enable more thorough testing, which helps in detecting faults that might otherwise remain unnoticed. They make it easier to diagnose and fix issues within the circuit by allowing direct access to internal signals. Additionally, test points help in monitoring and changing internal states, which is crucial for testing and debugging.

Types of Test Points

There are two main types of test points: control points and observation points. Control points allow for changing the internal states of the circuit. They are used to inject test signals into the circuit, enabling the testing of specific functions. Observation points provide access to internal signals for monitoring purposes: they allow in fact to capture and analyze the behavior of the circuit during testing.

Test Point Insertion and Optimization Process

The process of inserting and optimizing test points involves several key steps to ensure an effective testability while having a minimal impact on circuit performance and layout. Initially, critical points within the circuit that require access for testing purposes must be identified. This involves analyzing the circuit to pinpoint points that are crucial for testing. Once these points are determined, test points are inserted to provide access for observation and control.

The placement of test points is then improved using various techniques. Automated test point insertion methods determine the optimal placement of test points based on the circuit design and testing requirements. Fault simulation involves testing for potential faults to identify the most critical points for test point insertion. Design for Debug (DFD) approaches apply DFD principles to enhance

the testability and debug capabilities of the circuit. These methods balance the need for access with the effect on circuit performance and layout. Techniques like X-filling minimize the number of transitions during testing, thereby reducing power consumption and mitigating the increased power usage.

Together, these steps and techniques ensure that test points are effectively integrated into the design, providing the necessary access for observation and control while maintaining the circuit performance and layout integrity.

The information provided aboud the DFD in this section is derived from [10] and [11].

2.3.6 Memory BIST

MBIST is a DfT technique specifically designed to test memory components within an SoC. MBIST automates the process of generating test patterns and analyzing the responses, ensuring an exhaustive testing of the memory blocks. This technique is essential for detecting faults in memory components and ensuring their reliability. The architecture of MBIST typically includes a test pattern generator, a response analyzer, and control logic. The test pattern generator produces test vectors that are applied to the memory blocks, while the response analyzer compares the memory's output with the expected results to detect faults.

There are various types of MBIST, including programmable MBIST and deterministic MBIST. Programmable MBIST allows for the customization of test patterns, while deterministic MBIST uses predefined patterns to ensure thorough test coverage.
Several industries have successfully implemented MBIST in their designs. For example, semiconductor companies use MBIST to test memory components in microprocessors and other complex ICs, while automotive electronics manufacturers use MBIST to ensure the reliability of safety-critical systems. MBIST offers several advantages, including automated testing, improved test coverage, and lower testing costs. However, it also has limitations, such as increased design complexity and possible performance impact.

The information provided in this section is derived from [4].

2.4 Hardening Techniques

Hardening techniques are employed to enhance the reliability and fault tolerance of SoCs. These techniques are generally based on spatial on temporal redundancy, i.e., on the replication of logic or on the replication of computational tasks. The first category includes TMR and Error Correction Code (ECC). TMR involves triplicating critical components and using majority voting to determine the correct output, while ECC adds redundancy to data storage to detect and correct errors.

2.4.1 Triple Modular Redundancy

TMR is a fault-tolerant design technique that involves triplicating critical components and using a majority voting system to determine the correct output. This method ensures that even if one of the three components fails, the system can still produce the correct output based on the majority vote. TMR is widely used in safety-critical applications, such as aerospace and medical devices, where reliability is of paramount importance (see Figure 2.3).

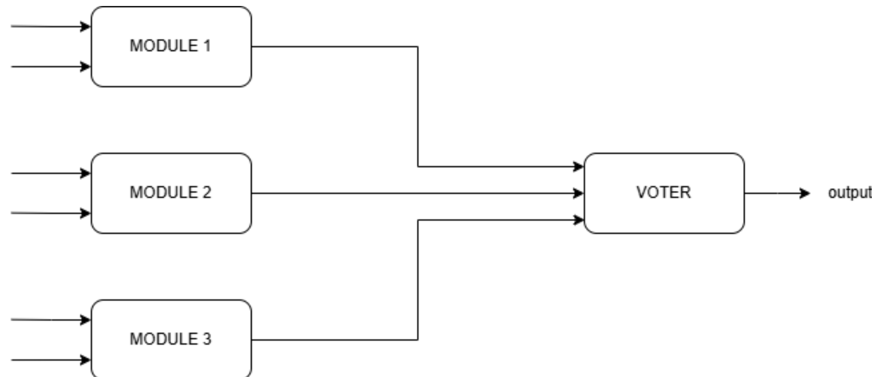

Figure 2.3: Triple Modular Redundancy (TMR) Architecture

2.4.2 Error Correction Code

ECC is a technique used to detect and correct errors in data storage and transmission. It works by adding redundancy to the data, thus allowing the system to identify and correct single-bit errors and detect double-bit errors. This technique is commonly used in memory systems, communication systems, and data storage devices to ensure data integrity and reliability.

In Hamming code for Single Error Correction, Double Error Detection (SEC-DED), specific bit positions are designated as either parity bits or data bits. The arrangement of these bits is shown in Table 2.2.

Bit Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit Type	P1	P2	D1	P3	D2	D3	D4	P4	D5	D6	D7	D8	D9	D10	D11

Table 2.2: Hamming Code (SEC-DED) Bit Positions: P denotes parity bits and D denotes data bits.

The parity bits are calculated based on the data bits they cover. The specific bit positions each parity bit covers and example calculations for these parity bits are shown in Table 2.3.

Parity Bit	Covers Bit Positions	Example Calculation
P1	1, 3, 5, 7, 9, 11, 13, 15	P1 = D1 ⊕ D2 ⊕ D4 ⊕ D5 ⊕ D7 ⊕ D9 ⊕ D11
P2	2, 3, 6, 7, 10, 11, 14, 15	P2 = D1 ⊕ D3 ⊕ D4 ⊕ D6 ⊕ D7 ⊕ D10 ⊕ D11
P3	4, 5, 6, 7, 12, 13, 14, 15	P3 = D2 ⊕ D3 ⊕ D4 ⊕ D8 ⊕ D9 ⊕ D10 ⊕ D11
P4	8, 9, 10, 11, 12, 13, 14, 15	P4 = D5 ⊕ D6 ⊕ D7 ⊕ D8 ⊕ D9 ⊕ D10 ⊕ D11

Table 2.3: Hamming Code (SEC-DED) Parity Bit Calculation: Example calculations for parity bits.

The informations provided in these last sections are from this book [12]. For further reading on hardening techniques, the reader can consult the following references for more comprehensive insights: [4, 13]

Many of the concepts and techniques discussed in this sections above are elaborated in the books [14] and [15].

2.5 Overview of System on Chip

A **SoC** is an IC that consolidates all the essential components of a computer or other electronic systems into a single chip. These components typically include a CPU, memory, input/output ports, and secondary storage, among others. SoCs are used in a wide array of devices, from smartphones and tablets to embedded systems and IoT devices.

The key components of an SoC are the CPU, which is the brain of the SoC responsible for executing instructions and performing calculations, the memory, which includes both volatile memory (RAM) and non-volatile memory (flash storage) for storing data and instructions, the input/output ports, which facilitate communication between the SoC and external devices, such as sensors, displays, and other peripherals. Furthermore, SoC may include other components too, such as GPU, integrated to handle rendering and image processing tasks in devices that require graphical output; Digital Signal Processors (DSPs) which are specialized for processing real-time signals, such as audio and video. Connectivity modules, including Wi-Fi, Bluetooth, GPS, and other communication interfaces.

SoCs offer several advantages. First, they significantly reduce the size of the device by integrating multiple components into a single chip. They are designed to be power-efficient, making them ideal for battery-operated devices. With components closely integrated, SoCs can offer high performance and faster data transfer rates. Additionally, manufacturing a single chip is often more cost-effective than producing multiple discrete components.

As previously noted, SoCs are found in many modern electronics and are used in mobile devices such as smartphones, tablets, and wearable technology. They are also used in embedded systems, including automotive electronics, industrial control systems, and home automation. Consumer electronics like smart TVs, gaming consoles, and digital cameras also utilize SoCs. Furthermore, SoCs are integral to IoT devices, including smart home devices, health monitoring systems, and environmental sensors.

For more detailed information on SoC, please refer to the Wikipedia article [16] and the book [17].

2.6 RISC-V Instruction Set Architecture

RISC-V is an open-source Instruction Set Architecture (ISA) that has gained significant traction in both the academic and industrial worlds due to its flexibility, scalability, and the benefits of being open-source. Developed at the University of California, Berkeley, RISC-V has evolved to become a robust and versatile ISA suitable for a wide range of applications.

2.6.1 Key Features of RISC-V

RISC-V key features include modularity, simplicity, efficiency, and extensibility. It is designed to be modular, allowing for different extensions (e.g., RV32I, RV64I, RV128I) to be added based on application requirements. This modularity enables designers to tailor the ISA to specific needs, enhancing performance and efficiency. The simplicity of the RISC-V ISA makes it easier to implement and verify, leading to more efficient hardware designs. Its clean and straightforward design reduces the complexity of the processor, which in turn can lead to lower power consumption and higher performance. Additionally, RISC-V's extensibility allows for custom instructions to be added without compromising the base ISA, which is particularly valuable for specialized applications that require unique processing capabilities.

2.6.2 Advantages of RISC-V

One of the primary advantages of RISC-V is its open-source nature. This reduces costs, and allows for customization of the ISA to meet specific application requirements. The open-source model also encourages collaboration and sharing within the community. RISC-V benefits from a growing ecosystem of tools, software, and community support. This includes compilers, simulators, development boards, and a wide range of software libraries, all of which facilitate development and adoption. Furthermore, RISC-V can be optimized for performance and power

efficiency, making it suitable for a wide range of applications from embedded systems to high-performance computing. So overall the ability to customize the ISA allows designers to achieve the desired balance between performance and power consumption.

2.6.3 Applications of RISC-V

RISC-V is widely used in embedded systems, including IoT devices, microcontrollers, and low-power applications. Its low power consumption and small footprint make it ideal for these applications. Additionally, RISC-V is being adopted in high-performance computing, including data centers and supercomputers. Its extensibility and performance capabilities make it a strong contender in this space. On the other side, RISC-V open-source nature and simplicity make it an ideal teaching tool in academic research and education. Many universities and research institutions use RISC-V to teach computer architecture and explore new research ideas.

2.6.4 RISC-V in SoC Design

RISC-V cores can be integrated into SoC designs, providing flexibility and customization options for various applications. This integration allows for the development of highly specialized and efficient SoCs. Several successful SoC designs based on RISC-V have demonstrated its capabilities in terms of performance, reliability, and testability. These case studies highlight the practical benefits of using RISC-V in modern SoC design.

2.6.5 Challenges and Future Directions

Despite its advantages, RISC-V faces some challenges, such as the need for a more mature ecosystem and potential fragmentation due to its extensibility. Addressing these challenges is crucial for the continued growth and adoption of RISC-V. However the future of RISC-V looks very promising, with ongoing research and development aimed at improving the ISA and expanding its applications. Emerging technologies and new use cases will continue to drive the evolution of RISC-V.

This section information are provided in this books [18] and [19]. For more detailed information on RISC-V, please refer to the Wikipedia article [20].

2.7 Ibex Overview

The Ibex core, developed by the PULP platform, is a small, efficient, and open-source RISC-V processor core designed for embedded systems and Internet of Things

(IoT) applications. It emphasizes energy efficiency and low power consumption while maintaining a flexible and configurable architecture. This section provides a detailed overview of the architecture, features, design considerations, and applications of the Ibex core.

2.7.1 Architecture

The Ibex core is a 32-bit RISC-V processor that implements the RV32IMC ISA. The core is designed with a two-stage pipeline, consisting of the Fetch Stage and the Execute Stage. In the Fetch Stage, instructions are fetched from memory, and the Program Counter (PC) is updated to point to the next instruction. The instruction memory interface supports both instruction and data fetches. In the Execute Stage, the fetched instruction is decoded and executed. The result is written back to the appropriate register or memory location, and this stage includes an Arithmetic Logic Unit (ALU) for performing operations.

The two-stage pipeline architecture simplifies the control logic and minimizes the power consumption by reducing the number of active components at any given time. This design choice also reduces the complexity of hazard detection and forwarding mechanisms, which are critical for maintaining high instruction throughput and minimizing pipeline stalls.

2.7.2 Features

The Ibex core includes several features that make it suitable for low-power and embedded applications. It supports the RV32IMC ISA, which includes the base integer instruction set (RV32I), multiplication and division instructions (M), and compressed instructions (C) for code size reduction. The two-stage pipeline simplifies the design and reduces power consumption, as well as the complexity of hazard detection and forwarding. The core can be configured with an optional hardware loop and a multiplier, in order to enhance the performance for certain applications. Specifically, Hardware loops reduce the overhead of loop control, and the optional multiplier can be included for performance-critical applications. The core also supports various low-power modes, including clock gating and power gating, and dynamic power management to reduce energy consumption. Debug support is provided through a debug module compliant with the RISC-V Debug Specification, supporting both Joint Test Action Group (JTAG) and serial wire debug (SWD) interfaces.

2.7.3 Design Considerations

The design of the Ibex core focuses on achieving a balance between performance, area, and power consumption. The core is optimized for low power consumption, making it ideal for battery-powered devices. Techniques such as clock gating and power gating are used to minimize energy usage. Moreover the core is designed to have a small silicon area, which is beneficial for cost-sensitive applications. It uses a minimalistic design approach in order to reduce the area without compromising essential features. Flexibility is another key aspect of the design, allowing the core to be configured with or without certain features, such as the hardware loop and multiplier, to meet specific application requirements. Configurable options enable the core to be tailored for different performance and power needs.

2.7.4 Applications

The Ibex core is suitable for a wide range of applications. Its low power consumption and small footprint make it ideal for IoT devices, including sensors, actuators, and other IoT components. The core energy efficiency is beneficial for wearable devices that require long battery life, such as fitness trackers, smartwatches, and health monitoring devices, as well as for embedded systems, where the core flexibility and low power modes make it suitable for various applications, including microcontrollers, automotive systems, and industrial automation.

2.8 OBI Protocol

2.8.1 Overview of OBI Protocol

The Open Bus Interface (OBI) protocol is a flexible and efficient bus protocol designed for communication between different components in a SoC. It is particularly suitable for low-power and high-performance applications. The OBI protocol supports a wide range of features that enhance its flexibility and efficiency. It supports multiple transfer types, including single and burst transfers, which allows for efficient data movement. The protocol also includes support for different data widths, ranging from 8-bit to 64-bit transfers, accommodating various application requirements. Additionally, the OBI protocol provides mechanisms for error detection and correction, ensuring reliable data transfers. The OBI protocol defines several transaction types to handle different communication scenarios. These include read and write transactions, which are used for standard data transfers between the core and peripherals. The protocol also supports atomic transactions, which ensure that a series of operations are completed without interruption, providing a mechanism for implementing critical sections in software.

The following figures illustrate various timing diagrams for different transaction types in the OBI protocol.

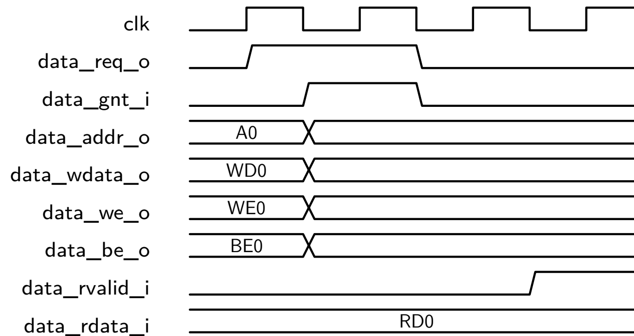

Figure 2.4: Basic Memory Transaction

Figure 2.4 shows a basic memory transaction where the core requests data, and the peripheral grants access and provides the requested data.

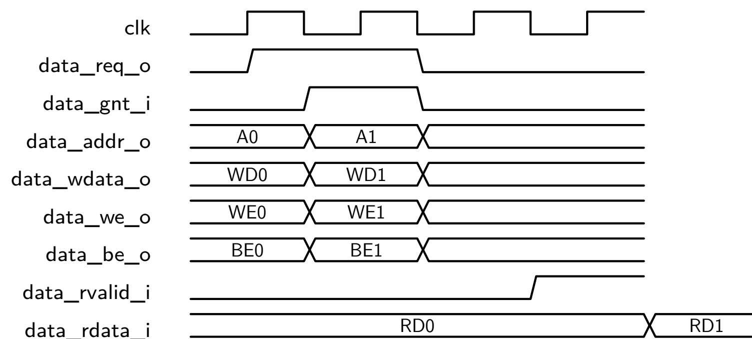

Figure 2.5: Back-to-back Memory Transactions

Figure 2.5 illustrates back-to-back memory transactions where multiple data requests are made in quick succession.

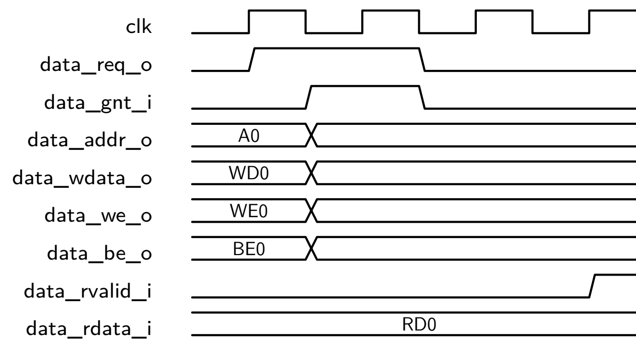

Figure 2.6: Slow Response Memory Transaction

Figure 2.6 depicts a slow response memory transaction where there is a delay in the peripheral providing the requested data.

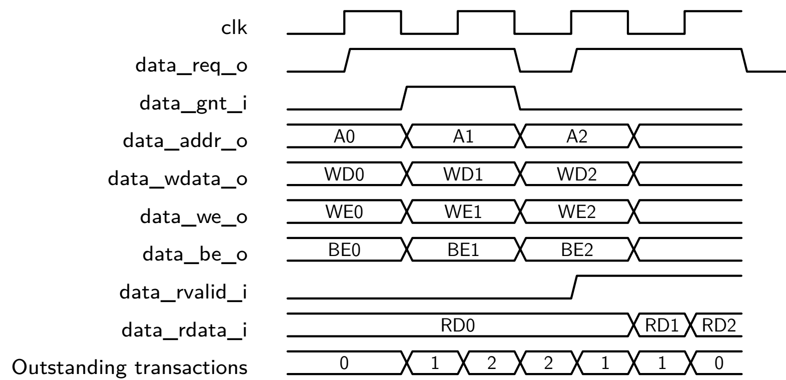

Figure 2.7: Multiple Outstanding Memory Transactions

Figure 2.7 shows multiple outstanding memory transactions where several requests are processed simultaneously, and the number of outstanding transactions is tracked.

Bus Arbitration

Bus arbitration in the OBI protocol is designed to be efficient and fair, ensuring that all components have access to the bus when needed. The protocol supports both fixed and dynamic priority schemes, allowing for flexible arbitration policies. In a fixed priority scheme, each component is assigned a fixed priority level, while in a dynamic priority scheme, the priority levels can change based on the current system state.

Low-Power Features

The OBI protocol includes several features aimed at reducing power consumption. It supports clock gating, which allows the clock signal to be disabled for certain components when they are not in use. This reduces dynamic power consumption by preventing unnecessary switching activity. The protocol also supports power gating, which can completely power down unused components, further reducing power consumption.

The information provided in this section is derived from [21, 22].

2.9 CVA6 Overview

The CVA6 core, formerly known as Ariane, is an advanced, open-source RISC-V processor core designed for high-performance applications. Developed by the PULP platform, it targets a wide range of applications, from embedded systems to high-performance computing. This section provides a detailed overview of the architecture, features, design considerations and applications of the CVA6 core.

2.9.1 Architecture

The CVA6 core is a 64-bit RISC-V processor that implements the RV64GC ISA. It features a six-stage, in-order pipeline, which includes the Fetch, Decode, Execute, Memory, Write-back, and Commit stages. This architecture allows for high instruction throughput and efficient execution of complex operations.

In the Fetch stage, instructions are fetched from memory, and the PC is updated to point to the next instruction. The Decode stage decodes the fetched instruction and prepares the necessary operands. The Execute stage performs the required arithmetic or logical operations using the ALU. The Memory stage handles memory access operations, while the Write-back stage writes the results back to the appropriate registers. Finally, the Commit stage ensures that instructions are retired in order, maintaining the architectural state of the processor.

2.9.2 Features

The CVA6 core includes several advanced features that make it suitable for high-performance applications. It supports the RV64GC ISA, which includes the base integer instruction set (RV64I), multiplication and division instructions (M), atomic instructions (A), floating-point instructions (F and D), and compressed instructions (C).

The core features a six-stage, in-order pipeline, which allows for high instruction throughput and efficient execution of complex operations. It also includes support

for hardware virtualization, enabling the core to run multiple operating systems or virtual machines simultaneously. Additionally, the CVA6 core supports various low-power modes, including clock gating and power gating, to reduce energy consumption.

Debug support is provided through a debug module compliant with the RISC-V Debug Specification, supporting both JTAG and serial wire debug interfaces. The core also includes support for hardware performance counters, which can be used to monitor and optimize the performance of applications.

2.9.3 Design Considerations

The design of the CVA6 core focuses on achieving a balance between performance, area, and power consumption. The core is optimized for high performance, making it suitable for applications that require high computational power. Techniques such as clock gating and power gating are used to minimize energy usage, making the core suitable for energy-sensitive applications.

The core is designed to have a moderate silicon area, balancing the need for high performance with the constraints of area and, thus cost. A modular design approach is taken, allowing the core to be easily integrated into larger SoCs. Flexibility is a key aspect of the design, allowing the core to be configured with or without certain features, such as hardware virtualization, to meet specific application requirements.

2.9.4 Applications

The CVA6 core is suitable for a wide range of applications. Its high performance and comprehensive instruction set make it ideal for high-performance computing applications, including scientific computing, data analytics, and machine learning. The core support for hardware virtualization makes this core suitable for use in data centers and cloud computing environments.

In embedded systems, the CVA6 flexibility and high performance make it suitable for applications such as automotive systems, industrial automation, and advanced robotics. The core low-power features also make it suitable for portable and energy-sensitive applications, such as mobile devices and wearable electronics.

The information contained in this section is derived from [23].

2.10 SRAM Overview

Static Random Access Memory (SRAM) is a type of semiconductor memory that uses bistable latching circuitry to store each bit. Unlike Dynamic RAM (DRAM), SRAM does not need to be periodically refreshed, which makes it faster and more

reliable for certain applications. This section provides a detailed overview of the architecture, features, design considerations, and applications of SRAM.

2.10.1 Architecture

The basic architecture of SRAM consists of an array of memory cells, each capable of storing one bit of data. Each SRAM cell is typically made up of six transistors (6T), although other configurations such as 4T and 8T cells also exist. The 6T cell configuration is the most common due to its balance between stability and area efficiency.

In a 6T SRAM cell, two cross-coupled inverters form a bistable circuit, and two access transistors control the read and write access to the cell. The cross-coupled inverters maintain the state of the cell, while the access transistors connect the cell to the bit lines during read and write operations. The word line controls the access transistors, enabling or disabling the connection to the bit lines. The structure of a typical 6T SRAM cell is illustrated in Figure 2.8.

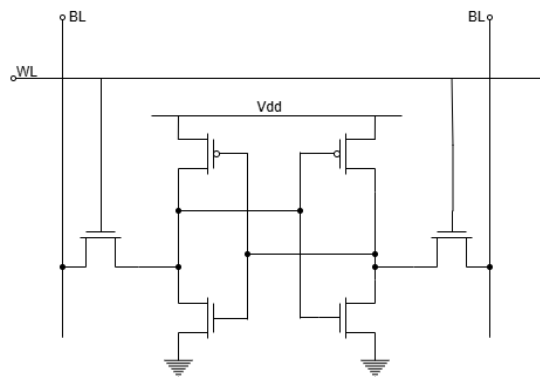

Figure 2.8: Schematic of a 6T SRAM cell.

Read Operation

During a read operation, the word line is activated, turning on the access transistors and connecting the SRAM cell to the bit lines. The stored value in the cell is then transferred to the bit lines, where it can be sensed and read by the sense amplifiers. The differential nature of the bit lines helps in quickly determining the stored value by comparing the voltage levels.

Write Operation

During a write operation, the word line is activated, and the desired value is driven onto the bit lines. The access transistors connect the bit lines to the SRAM cell,

allowing the new value to overwrite the existing value stored in the cell. The cross-coupled inverters then latch onto the new value, maintaining it until the next write operation.

2.10.2 Features

SRAM offers several features that make it suitable for high-speed and low-power applications. It provides fast access times, typically in the order of nanoseconds, making it ideal for use in cache memory and other performance-critical applications. SRAM also has low power consumption in standby mode, as it does not require periodic refreshing like DRAM.

The volatile nature of SRAM means that it retains data as long as power is supplied, making it suitable for applications that require persistent data storage without the need for refresh cycles. Additionally, SRAM cells are relatively simple in design, allowing for high-density integration in semiconductor devices.

2.10.3 Design Considerations

The design of SRAM involves several considerations to achieve a balance between performance, area, and power consumption. The choice of cell configuration (e.g., 6T, 4T, 8T) impacts the stability, area efficiency, and power consumption of the memory.

Leakage current is a critical factor in SRAM design, as it affects both power consumption and data retention. Techniques such as power gating and body biasing can be employed to minimize leakage current and improve power efficiency. The design of the sense amplifiers and bit lines also plays a crucial role in determining the read and write performance of SRAM.

Process variations and manufacturing defects can impact the yield and reliability of SRAM. ECC and redundancy techniques can be used to enhance the reliability and fault tolerance of SRAM arrays.

2.10.4 Applications

SRAM is widely used in various applications due to its high speed and low power consumption. In microprocessors, SRAM is commonly used as cache memory to store frequently accessed data and instructions, reducing the latency of memory access and improving overall system performance. SRAM is also used in embedded systems, where its fast access times and low power consumption are critical for real-time processing and battery-powered devices.

In networking equipment, instead SRAM is used for packet buffering and routing tables, where high-speed memory access is essential for maintaining data throughput

and network performance. SRAM is also used in graphics processing units (GPUs) for storing frame buffers and texture data, enabling high-performance rendering and image processing.

The information provided in this section is derived from [24] and from the book [25].

2.10.5 Faults in Memory and March Tests

While SRAM offers high speed and reliability, it is not immune to faults. Memory faults can occur in digital memories, affecting the integrity of stored data. Common faults include stuck-at faults (where a bit is stuck at 0 or 1), transition faults (where a bit fails to change state correctly), and coupling faults (where the state change of one bit affects another bit).

To detect and diagnose these faults, various testing methods are used, including **March Tests**. March Tests are sequences of read and write operations performed on each memory cell to identify faults. Each operation in a March Test is executed in a specific order and can include actions like writing 0, reading 0, writing 1, and reading 1. An example of a March Test is the **March C-Test**, which consists of the following steps:

1. Write 0 to all cells.
2. Read 0 from all cells.
3. Write 1 to all cells.
4. Read 1 from all cells.
5. Write 0 to all cells.
6. Read 0 from all cells.

These tests are designed to be efficient and cover a wide range of faults, ensuring that the memory operates correctly under various conditions.

The information for this section are provided from this website [26]

Chapter 3

Approach

3.1 Comprehensive Approach to Enhancing SoC Reliability and Testability

This chapter presents a complete approach to improving the reliability, testability, and availability of modern SoC designs. The proposed solution combines advanced DfT techniques, hardening methods, and ECC techniques. These methods help achieve high test coverage, reduced test time, better reliability, and increased availability. The chapter begins by outlining the general solution, followed by a detailed explanation of the conceptual framework that supports the approach. The applicability of the solution to different scenarios is then discussed, highlighting its flexibility. Finally, the advantages and limitations of the approach are examined, providing a fair view of its potential impact.

3.2 Conceptual Framework

The conceptual framework of this approach is built on widely recognized theories and methods. The framework combines principles from systems theory and engineering to create a complete solution. The key components of the framework include DfT and hardening techniques.

3.3 Design for Testability

DfT techniques are crucial for enhancing the testability of SoCs. This approach includes several advanced DfT techniques, such as LBIST, scan chains, test points, and MBIST. These techniques help detect and diagnose faults within the SoC, ensuring complete test coverage and reducing test time.

3.3.1 General Method to Implement DfT

To insert DfT into an SoC, the process begins by defining the specific test requirements and goals, such as test coverage, test time, and test cost. The SoC is then divided into smaller, manageable modules or blocks, which helps in isolating faults and simplifies testing. Next, standard test access mechanisms like JTAG (IEEE 1149.1) are implemented for boundary scan testing, ensuring that these mechanisms provide access to internal nodes and facilitate external control and observation [27].

3.3.2 Implementation of Scan Chains

Scan chains are integrated into the design to enable the shift-in and shift-out of test vectors. Tools are used to automatically insert scan chains and optimize their length and number.

Observability and controllability are enhanced by ensuring that internal signals and states can be observed and controlled during testing: this is achieved using multiplexers, test points, and additional logic, which facilitate the monitoring and manipulation of internal states.

Example: Scan Chains Implementation

Consider a SoC design with multiple functional blocks such as CPU, memory, and I/O controllers. Each block is equipped with scan chains that allow test patterns to be shifted in and out: this enables an exhaustive testing of each block independently, ensuring that faults can be detected and diagnosed efficiently [28].

3.3.3 Implementation of Scan Compression

Scan compression techniques are applied to reduce the volume of test data and test time. These techniques involve compressing the test patterns before they are loaded into the scan chains and decompressing them on-chip. This approach significantly reduces the amount of test data that needs to be stored and transferred, leading to faster and more efficient testing.

The process of implementing scan compression begins with selecting appropriate compression algorithms and tools: these generate compressed test patterns, which are then decompressed on-chip using specialized decompression hardware (which works with the existing scan chains ensuring that the test patterns are correctly applied to the SoC).

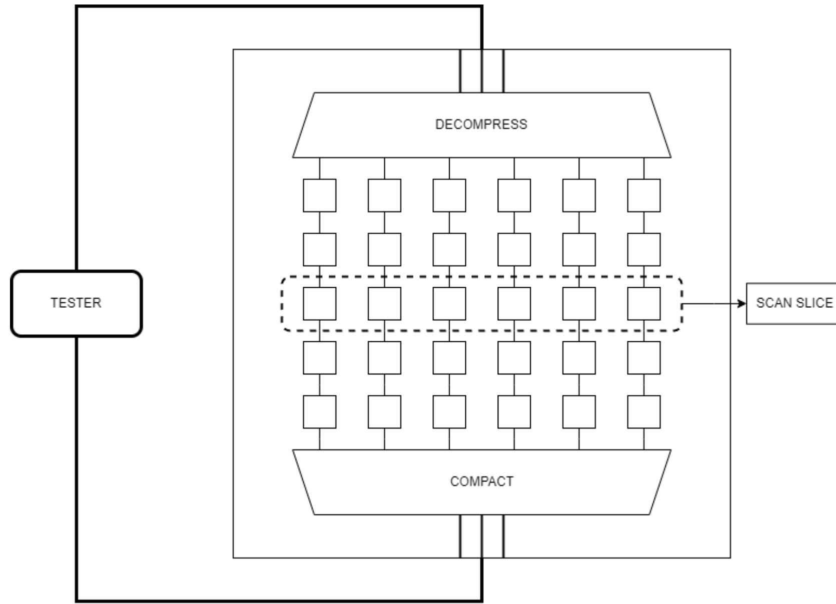

Figure 3.1: Example of Scan Compression Architecture

Consider a SoC design with multiple scan chains. As shown in Figure 3.1, scan compression involves loading compressed test patterns decompressed on-chip using dedicated hardware, which distributes the test data efficiently across the scan chains. By reducing the data volume and the number of cycles required, scan compression leads to shorter testing times and a reduction in storage requirements.

3.3.4 Implementation of Memory BIST

In a memory BIST implementation, March tests are used to detect faults in SRAM cells. The BIST controller generates test patterns and compares the output with expected results. Any discrepancy indicates the presence of faults, which are then logged for further analysis [29].

Test Type	Test Coverage	Test Time
March Test	High	Moderate
Random Test	Moderate	Low
Pattern Test	Low	High

Table 3.1: Comparison of Different Memory BIST Tests: This table compares various memory BIST test types in terms of test coverage and test time.

Test compression techniques are applied to reduce the volume of test data and test time, and tools are used to generate compressed test patterns and decompress them on-chip. Fault simulation is conducted to evaluate the effectiveness of the test structures, and test coverage is analyzed to ensure that the test goals are met. The DfT implementation is validated through simulation and emulation, ensuring that the DfT logic does not interfere with the functional operation of the SoC. Finally, test patterns for manufacturing tests, including stuck-at, transition, and path delay faults, are generated, and test programs for Automatic Test Equipment (ATE) are developed to execute the test patterns.

3.3.5 Implementation of Logic BIST

As mentioned aforehand, LBIST is a DfT technique that allows the SoC to test its own logic circuitry without the need for external test equipment: it generates test patterns internally and compares the results with expected values, in order to identify faults [30].

Example: In a SoC design, LBIST can be used to test the ALU. The LBIST controller generates a series of test patterns to exercise various operations of the ALU, such as addition, subtraction, multiplication, and division. The results are compared with expected values to identify any discrepancies. This ensures that the ALU is functioning correctly and can detect faults that might otherwise go unnoticed.

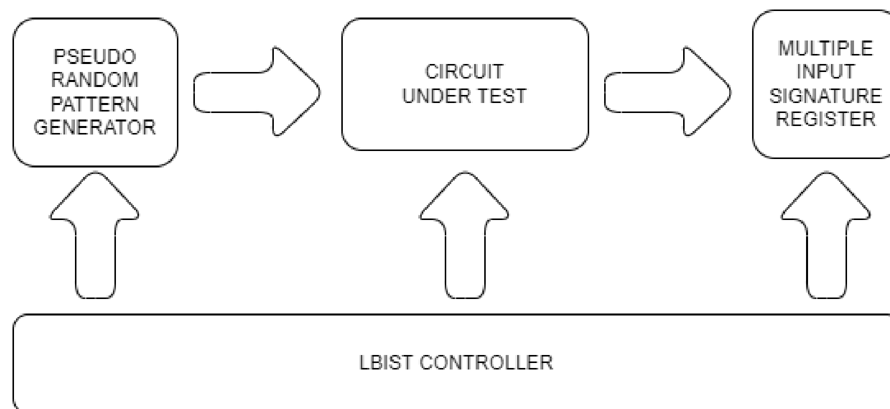

Figure 3.2: General LBIST Implementation: This figure illustrates a versatile LBIST setup, applicable across various design architectures for effective logic self-testing.

3.3.6 Implementation of Test Points

Test points are strategically placed within the SoC to enhance observability and controllability. They allow test signals to be injected and observed at various points in the circuit, facilitating fault detection and diagnosis.

Example: In a SoC design, test points can be placed at the input and output of a critical data path, such as a high-speed data bus. During testing, signals can be injected at the input test point and observed at the output test point. This allows for the detection of any faults that may occur along the data path.

3.4 Hardening Techniques

Hardening techniques, such as TMR and ECC, are employed to enhance the reliability and fault tolerance of the SoC. In particular, TMR achieves fault tolerance by triplicating critical components and applying majority voting to mask faults, while ECC adds redundancy to data storage to detect and correct errors [31]. Additional techniques provide further resilience, as summarized in Table 3.2.

3.4.1 General Method to Implement Hardening Techniques

To harden an SoC, the process begins by defining the security requirements, including threat models, attack surfaces, and rules. The SoC design is then divided into security zones to limit the impact of potential breaches. Secure access mechanisms are implemented to protect test access mechanisms against unauthorized access, possibly by using secure JTAG. Moreover secure scan techniques are used to prevent scan chain-based attacks [32].

3.4.2 Implementation of ECC in Memory Controllers

In a secure SoC design, critical data paths are protected using ECC. For instance, in a memory controller, ECC can detect and correct single-bit errors, which ensures data integrity even in the presence of faults [33]. Figure 3.3 illustrates an ECC implementation in a memory system.

Security BIST is added to implement security checks within BIST to detect interference or unauthorized access. By ensuring that monitoring mechanisms are secure, observability and controllability are enhanced. Secure compression techniques are integrated to ensure that compressed test data is encrypted and protected. Security testing is performed to identify and reduce vulnerabilities, and security features are validated to ensure they are effective against identified threats. Finally, security test patterns are developed specifically for security features and vulnerabilities.

Hardening Technique	Description
Triple Modular Redundancy	Triplicates critical components within the design and applies majority voting to mask faults. Commonly used to enhance fault tolerance in critical logic.
Error Correction Code	Adds redundancy within data paths or memory cells to detect and correct single-bit or multi-bit errors, protecting data integrity.
Hardened Interconnects	Reinforces communication pathways by designing fault-tolerant interconnect architectures to reduce susceptibility to crosstalk and environmental interference.
Parity Checks	Integrates parity bits into data paths for quick error detection. Often used in memory designs for low-overhead error checking.
Fault Isolation Zones	Partitions the design into isolated fault zones to limit the impact of a fault within a region, enhancing fault containment.

Table 3.2: Design-Based Hardening Techniques for SoC Reliability and Fault Tolerance

Systems theory provides a complete understanding of the complex interactions within the SoC. By viewing the SoC as a system of interconnected components, key variables and their relationships can be identified. This overall view allows for the design of interventions that address the main reasons for faults rather than just the symptoms.

3.5 Applicability to Various Contexts

One of the strengths of this approach is its adaptability to different contexts. The framework is designed to be flexible, allowing it to be customized to meet the specific needs of various scenarios. In the following paragraphs, a couple of contexts in which this solution can be applied are outlined:

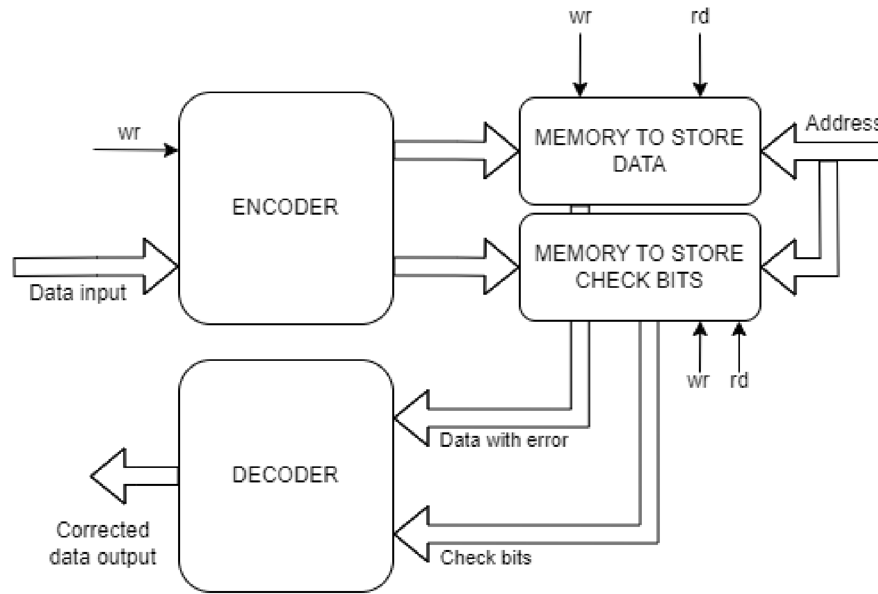

Figure 3.3: ECC Implementation in Memory systems: This figure shows the application of ECC in memory systems to ensure data integrity.

Automotive Sector

In the automotive sector, this approach can be used to improve the reliability of electronic control units (ECUs) and other critical components by ensuring complete testing and fault tolerance. By integrating DfT techniques and hardening methods, the reliability of critical components can be enhanced, reducing the risk of system failures and improving vehicle safety [34].

Example: In an autonomous vehicle, DfT techniques can be used to test the sensor processing unit, while hardening methods ensure that the vehicle continues to function correctly even if some components fail.

Aerospace Sector

In the aerospace sector, this solution can be applied to ensure the reliability and security of avionics systems: by incorporating DfT techniques and ECC, errors in critical data can be detected and corrected, reducing the risk of system failures and enhancing the overall safety of aerospace systems [35].

Example: In an aircraft's flight control system, ECC can be used to protect critical data paths, ensuring that any errors introduced during data transmission or storage are detected and corrected, maintaining the integrity of flight control operations.

3.6 Flowchart of the Development Process

The following flowchart 3.4 illustrates the steps involved in developing a system with DfT techniques and hardening techniques:

3.6.1 Explanation of Each Step

1. **Define System Requirements**:
    - Identify and document the specific requirements for the system, including performance, reliability, and testability criteria.
    - Gather input from stakeholders and define the scope of the project.
2. **System Design**:
    - Develop the overall architecture of the system, including the selection of components and interfaces.
    - Create detailed design specifications and schematics.
3. **Implement DfT Techniques**:
    - Integrate DfT techniques into the system design to enhance testability.
    - Techniques include LBIST, Scan Chains, Test Points, and MBIST.
    - Ensure that these techniques are incorporated early in the design phase to facilitate the testing phase.
4. **Implement Hardening Techniques**:
    - Apply hardening techniques to improve the reliability and fault tolerance of the system.
    - Techniques include TMR and ECC.
    - These help the system to operate correctly even in the presence of faults.
5. **Verification and Testing**:
    - Conduct comprehensive verification and testing of the design to ensure that it meets the specified requirements.
    - Use simulation tools to validate the functionality and performance of the system.
    - Perform fault simulation and analyze test coverage to fully test the device.
6. **Integration**:

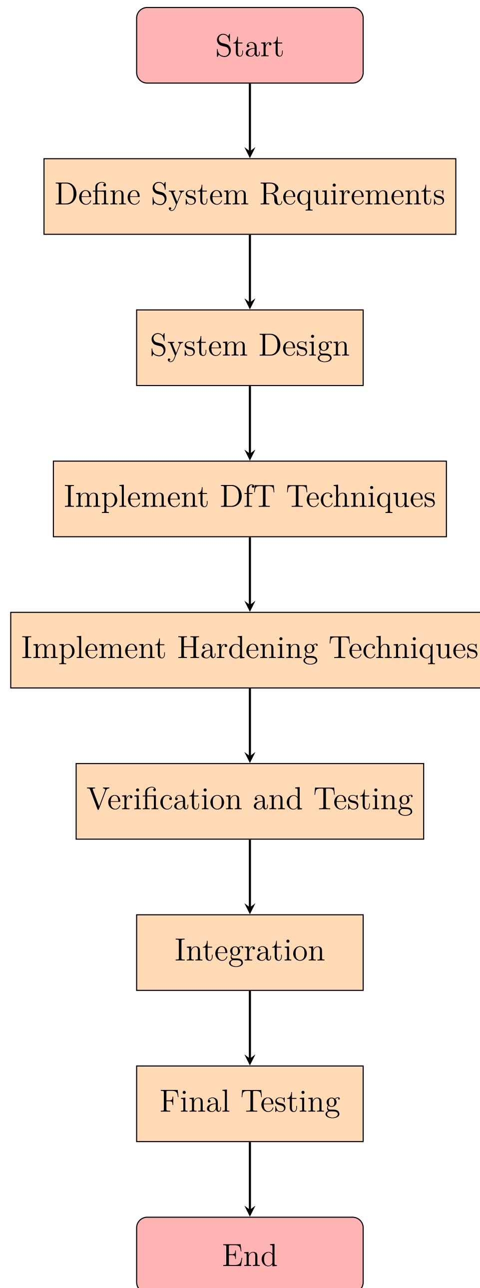

Figure 3.4: Flowchart of the Development Process with DfT and Hardening Techniques

- Integrate all components and modules of the system, ensuring that they work together seamlessly.
- Address any compatibility issues and verify that the system operates as

intended.

7. **Final Testing**:
   - Perform final testing of the integrated system to validate its overall functionality and performance.
   - Conduct stress testing, reliability testing, and performance testing to ensure that the system meets all requirements.
8. **End**: The process concludes with the successful development of the system, ready for deployment or further stages such as production.

Chapter 4

Implementation

4.1 Overview

This section provides an overview of the overall system architecture, setting the context for the subsequent detailed discussions on various components and techniques used in the design.

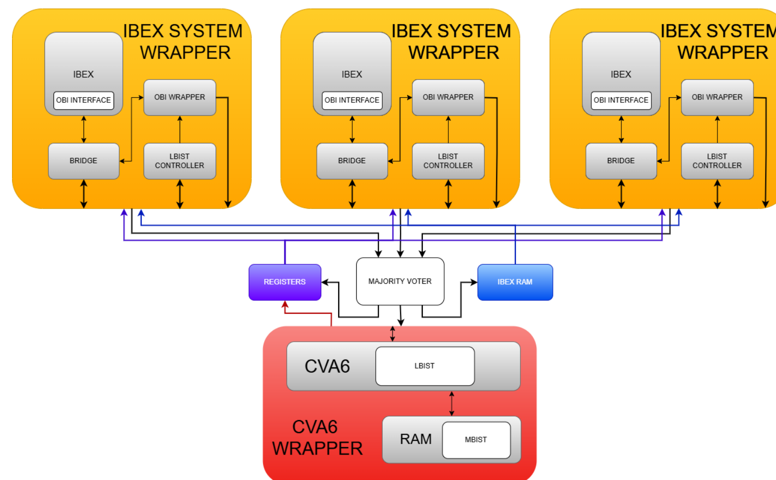

Figure 4.1: Overall System Architecture

Figure 4.1 illustrates the main components of the system. The architecture consists of three IBEX system wrappers, each containing an IBEX core, OBI interface, OBI wrapper, LBIST controller, and a bridge. These wrappers are connected to a majority voter, which ensures the reliability of the system by comparing the outputs from the three IBEX cores. The majority voter is also connected to the IBEX RAM and registers, which store the necessary data for

processing. Additionally, the architecture includes a CVA6 wrapper, which contains the CVA6 core, LBIST, RAM, and MBIST. This modular design makes the system scalable, maintainable, and capable of handling various processing tasks efficiently.

4.2 Methodology

4.2.1 Interface Definition

The interface definition is a crucial step in the implementation phase. For this project, it was decided to use the OBI already present in the Ibex processor. This involves specifying the communication protocols and data exchange formats between different components of the SoC. Among the others, the most noteworthy aspects include:

- establishing consistent signal naming conventions to ensure clarity and avoid conflicts;
- choosing appropriate communication protocols based on performance and compatibility requirements;
- defining timing specifications for signal transitions and data transfers to ensure synchronization between components.

4.2.2 System Wrapper Design

The system wrapper integrates the Ibex processor, bridge component, control module, and OBI wrapper component. This wrapper manages the communication between the processor, memory, and peripherals, guaranteeing reliable and fault-tolerant operation.

The system wrapper includes interfaces to the Ibex processor, RAM, and a register file. It also interfaces with an LBIST controller for the built-in self-test functionality. The wrapper handles data and instruction requests, routing them to the appropriate components and managing the responses.

The Ibex processor is instantiated within the system wrapper and connected to the bridge module. The bridge module routes requests to either the RAM or the OBI wrapper based on the address range. The OBI wrapper interfaces with the register file, handling read and write operations. The LBIST controller manages the built-in self-test operations and interfaces with the register file.

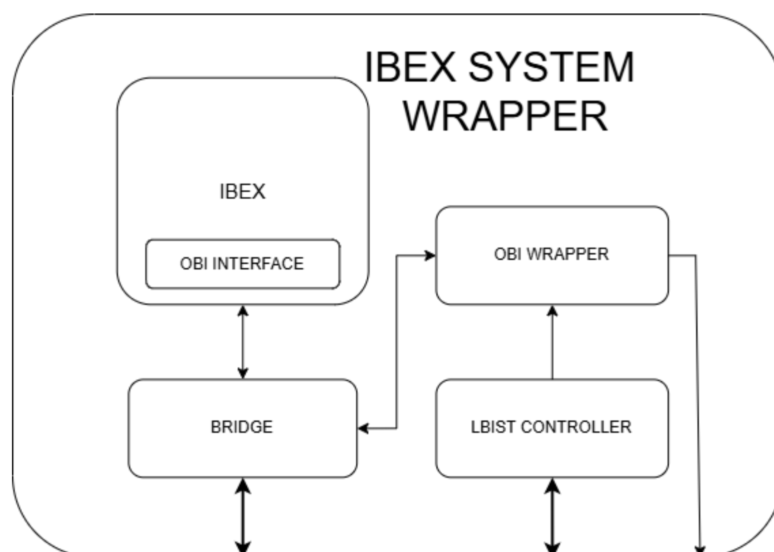

Figure 4.2: System Wrapper Design

The design of the system wrapper is illustrated in Figure 4.2.

4.2.3 Triple Modular Redundancy

As previously mentioned, TMR enhances system reliability and fault tolerance by triplicating critical components and using majority voting to determine the correct output. The TMR module thus helps preventing possible malfunctions in the presence of faults.

The `TMR` module takes three identical input signals and produces an output based on majority voting. If all three inputs are equal, the output is set to that value. If two inputs match, the output follows the matching inputs. If all inputs differ, the output defaults to the first input.

The design of the TMR architecture is illustrated in Figure 4.3. In this figure, the critical components that are triplicated include the IBEX System Wrapper (which contains the IBEX core), OBI Interface, OBI Wrapper, LBIST Controller, and Bridge. Each of these components is duplicated three times to form three separate IBEX System Wrappers. The outputs from these three IBEX System Wrappers are then fed into a majority voter module, which determines the final output based on the majority voting logic. Additionally, the IBEX RAM and Registers are also included in the process to ensure data integrity and consistency across the triplicated systems.

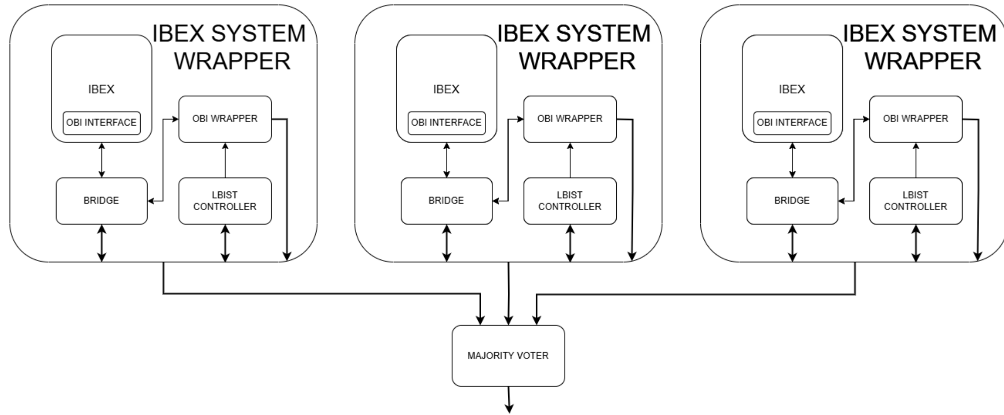

Figure 4.3: Triple Modular Redundancy System Architecture

4.2.4 Bridge Component

The bridge component between the Ibex processor and the peripherals (RAM and register file for programming the LBIST) includes arbitration to handle the peripherals. So, in essence, this component ensures that data requests and responses are correctly routed between the processor and the memory or register file.

The `bridge` module defines an address range for the OBI wrapper, with `OBI_ADDR_LO` set to `32'h00040084` and `OBI_ADDR_HI` set to `32'h000400FF`: these parameters help determine whether a request should be directed to the RAM or the OBI wrapper.

The module uses an arbiter state machine to manage the routing of requests. The state machine has three states: `IDLE`, `SERVICING_RAM`, and `SERVICING_OBI`. The current state is stored in the `arbiter_state` variable and determines the destination of the current request based on the address range and system state.

In the `IDLE` state, if a data request is made and the address is outside the OBI range, the state transitions to `SERVICING_RAM`. If the address is within the OBI range, the state transitions to `SERVICING_OBI`. In the `SERVICING_RAM` state, the state machine waits for the RAM transaction to complete, indicated by `data_rvalid_ram`, and then transitions back to `IDLE`.
Similarly, in the `SERVICING_OBI` state, the state machine waits for the OBI transaction to complete, indicated by `data_rvalid_obi`, and then transitions back to `IDLE`.

The output logic assigns the appropriate signals to RAM or OBI based on the current state of the arbiter. The response path multiplexing logic ensures that the correct response signals are sent back to the IBEX interface based on the current state of the arbiter. The instruction path is instead always handled by RAM.

The `bridge` module effectively manages the routing of data and instruction

requests between the IBEX core, RAM, and an OBI wrapper. By using an arbiter state machine, it determines the appropriate destination for each request based on the address range and the current state. The response signals are multiplexed to ensure the correct data is sent back to the IBEX interface.

The interface description of the bridge component is illustrated in Listing 4.1.

Listing 4.1: Bridge Component Interface

```
module bridge (
    input logic clk, rst_n,

    // Interface to IBEX
    input logic data_req_dut,
    input logic data_we_dut,
    input logic [3:0] data_be_dut,
    input logic [31:0] data_addr_dut,
    input logic [31:0] data_wdata_dut,
    input logic [6:0] data_wdata_intg_dut,
    input logic instr_req_dut,
    input logic [31:0] instr_addr_dut,

    output logic data_gnt_dut,
    output logic data_rvalid_dut,
    output logic [31:0] data_rdata_dut,
    output logic [6:0] data_rdata_intg_dut,
    output logic data_err_dut,
    output logic exit_success_dut,
    output logic instr_gnt_dut,
    output logic instr_rvalid_dut,
    output logic [31:0] instr_rdata_dut,
    output logic [6:0] instr_rdata_intg_dut,
    output logic instr_err_dut,

    // Interface to RAM
    output logic data_req_ram,
    output logic data_we_ram,
    output logic [3:0] data_be_ram,
    output logic [31:0] data_addr_ram,
    output logic [31:0] data_wdata_ram,
    output logic [6:0] data_wdata_intg_ram,
    output logic instr_req_ram,
    output logic [31:0] instr_addr_ram,

    input logic data_gnt_ram,
    input logic data_rvalid_ram,
    input logic [31:0] data_rdata_ram,
    input logic [6:0] data_rdata_intg_ram,
    input logic data_err_ram,
    input logic exit_success_ram,
```

```
    input logic instr_gnt_ram,
    input logic instr_rvalid_ram,
    input logic [31:0] instr_rdata_ram,
    input logic [6:0] instr_rdata_intg_ram,
    input logic instr_err_ram,

    // Interface to OBI wrapper
    output logic data_req_obi,
    output logic [31:0] data_addr_obi,
    output logic data_we_obi,
    output logic [3:0] data_be_obi,
    output logic [31:0] data_wdata_obi,
    output logic [6:0] data_wdata_intg_obi,

    input logic [31:0] data_rdata_obi,
    input logic data_rvalid_obi,
    input logic data_gnt_obi,
    input logic [6:0] data_rdata_intg_obi,
    input logic data_err_obi,
    input logic exit_success_obi
);
endmodule
```

4.2.5 OBI Wrapper Component

The OBI wrapper component is used to connect the bridge to the register file for the programmable LBIST. This component facilitates the communication between the bridge and the register file, ensuring that data integrity is maintained.

The `obi_wrapper` module is responsible for handling data requests and responses between the OBI interface and the register file. It includes an arbiter state machine to manage read and write operations and ensures data integrity through various signals.

The module defines an address width of 7 bits and a data width of 32 bits. It operates in different states: `IDLE`, `WRITE_READ`, `WRITE_DATA`, and `READ_DATA`. In the `IDLE` state, the system waits for a data request. Upon receiving a request, it transitions to the `WRITE_READ` state to determine whether to perform a write or read operation. The `WRITE_DATA` and `READ_DATA` states handle the respective operations and transition back to `IDLE` once complete.

The module ensures data integrity by using byte enable signals to selectively write or read specific bytes. It also includes default assignments for data integrity output signals and error indications.

The interface of the OBI_wrapper component is shown in Listing 4.2.

Listing 4.2: OBI Wrapper Component Interface

```
module obi_wrapper #(
    parameter ADDR_WIDTH = 7,
    parameter DATA_WIDTH = 32
)(
    input logic clk,
    input logic rst_n,

    // OBI port signals from Master to Slave
    input logic data_req_i,
    input logic [DATA_WIDTH-1:0] data_addr_i,
    input logic data_we_i,
    input logic [DATA_WIDTH/8-1:0] data_be_i,
    input logic [DATA_WIDTH-1:0] data_wdata_i,

    // OBI port signals from Slave to Master
    output logic [DATA_WIDTH-1:0] data_rdata_o,
    output logic data_rvalid_o,
    output logic data_gnt_o,

    // Data integrity signals
    input  logic [6:0] data_wdata_intg_i,
    output logic [6:0] data_rdata_intg_o,
    output logic data_err_o,
    output logic exit_success,

    // Register file signals
    output logic we1,
    output logic [ADDR_WIDTH-1:0] waddr1,
    output logic [ADDR_WIDTH-1:0] raddr1,
    output logic [DATA_WIDTH-1:0] wdata1,
    input logic [DATA_WIDTH-1:0] rdata1
);
endmodule
```

4.2.6 Register File

The register file is used to store the configuration and status information for the LBIST. It interacts with the OBI wrapper component to facilitate the read and the write operations. Moreover, the register file includes Hamming code encoders and decoders in order to ensure data integrity.

The `register_file` module defines an address width of 7 bits and a main data width of 32 bits, with an additional 7 bits for data integrity. It supports two write ports and two read ports, allowing simultaneous read and write operations. It is also worth noticing that the module includes logic to handle write-write conflicts by prioritizing port 1.

The module operates synchronously with the clock signal and resets all registers to 0 on reset. It uses address translation to map the input addresses to the corresponding memory locations. The read and write operations are performed based on the write enable signals, and the module ensures data integrity by forwarding data from the write ports during read operations when necessary.

The interface of the register file component is shown in Listing 4.3.

Listing 4.3: Register File Component Interface

```
module register_file #(
    parameter ADDR_WIDTH = 7,
    parameter MAIN_DATA_WIDTH = 32,
    parameter ADDITIONAL_WIDTH = 7
)(
    input logic clk,
    input logic rst_n,
    input logic we1,
    input logic we2,
    input logic [ADDR_WIDTH-1:0] waddr1,
    input logic [ADDR_WIDTH-1:0] waddr2,
    input logic [ADDR_WIDTH-1:0] raddr1,
    input logic [ADDR_WIDTH-1:0] raddr2,
    input logic [MAIN_DATA_WIDTH-1:0] wdata1,
    input logic [MAIN_DATA_WIDTH-1:0] wdata2,
    input logic [ADDITIONAL_WIDTH-1:0] wdata1_intg,
    input logic [ADDITIONAL_WIDTH-1:0] wdata2_intg,
    output logic [MAIN_DATA_WIDTH-1:0] rdata1,
    output logic [MAIN_DATA_WIDTH-1:0] rdata2,
    output logic [ADDITIONAL_WIDTH-1:0] rdata1_intg,
    output logic [ADDITIONAL_WIDTH-1:0] rdata2_intg,
    input logic request2,
    output logic valid_read_o,
    output logic valid_write_o
);
endmodule
```

4.2.7 LBIST Control Module

The control module takes the input from the register file to write in the LBIST and the output of the LBIST to write in the register file. This module manages the configuration and control signals for the LBIST, ensuring that the it operates correctly.

The `lbist_controller` module is responsible for managing the LBIST operations. It generates control signals and handles the state transitions required for the LBIST process. The module includes an internal state machine with states such as:

- IDLE

- READ_START
- READ_LBIST_EN
- WRITE_COMPLETE
- READ_SEED
- READ_SIGNATURE
- READ_PATTERN_COUNT
- READ_SHIFT_LENGTH
- READ_CAPTURE_CYCLE_ENABLE
- WAIT_FOR_STATUS
- RUN
- PASS
- FAIL

The module reads and writes various parameters such as the seed, signature, pattern count, and shift length. It also controls the capture cycle enable signal and monitors the status signals to determine the operation outcome. The state machine transitions between states based on the input signals and the current state, correctly organising the LBIST process.

The interface of the LBIST controller component is shown in Listing 4.4.

Listing 4.4: LBIST Controller Component Interface

```
module lbist_controller (
    input logic clk,
    input logic rst_n,
    output logic start,
    output logic lbist_en,
    input logic status_0,
    input logic status_1,
    output logic [96:0] SEED,
    output logic [29:0] SIGNATURE,
    output logic [19:0] PATTERN_COUNT,
    output logic [9:0] SHIFT_LENGTH,
    output logic CAPTURE_CYCLE_ENABLE,
    output logic we,
    output logic [6:0] waddr,
    output logic [6:0] raddr,
    output logic [31:0] wdata,
```

```
    input logic [31:0] rdata,
    output logic request,
    input logic valid_read,
    input logic valid_write,
    output logic TM,
    output logic LBIST_TestMode,
    output logic SCAN_COMPRESSION_ENABLE
);
endmodule
```

4.2.8 SEC-DED Hamming Code Encoder and Decoder

The SEC-DED (Single Error Correction, Double Error Detection) Hamming code encoder and decoder are used at the input and output of the register file to ensure data integrity. These components detect and correct single-bit errors and detect double-bit errors in the data. The `sec_ded_encoder_32` module takes a 32-bit input data and produces a 32-bit output data along with a 7-bit parity output. The input data is passed directly to the output, while the parity bits are calculated based on the input data. The parity bits are used to detect and correct errors in the data. The `sec_ded_decoder_32` module takes a 32-bit input data and a 7-bit parity input. It produces a 32-bit corrected data output, an error detected flag, and an error corrected flag. The module calculates syndrome bits to determine if there is an error in the data. If a single-bit error is detected, it corrects the error and sets the error corrected flag. If a double-bit error is detected, it sets the error detected flag. The ECC of the Hamming code is also used for both the transmission between the OBI interface and the OBI wrapper, and the writing of the data into the memory. this mechanism ensures that data integrity is maintained throughout the entire data path, from the OBI interface through the OBI wrapper, and into the memory.

The system configuration using the SEC-DED Hamming code encoder and decoder is illustrated in Figure 4.4.

4.2.9 Testbench

The testbench is used to verify the functionality of the entire system, including the system wrapper, bridge component, OBI wrapper component, control module, register file, TMR module, SEC-DED Hamming code encoder, and SEC-DED Hamming code decoder. The testbench simulates the operation of the system and checks that all components function correctly.

The testbench, defined in the file `tb_top.sv`, includes the following key elements:

- **Clock Generation**: A clock signal is generated with a specified period.

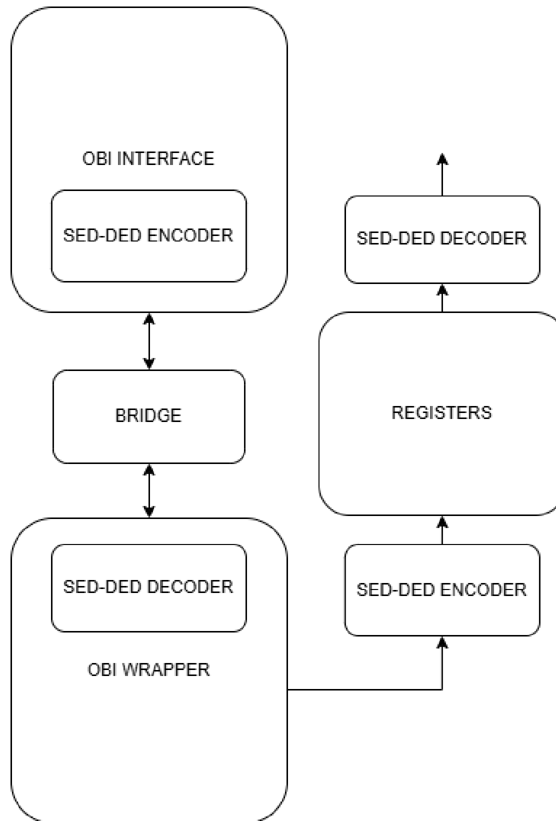

Figure 4.4: SEC-DED System architecture

- **Reset Activation**: The reset signal is activated for a defined number of clock cycles.
- **VCD Dump**: A VCD (Value Change Dump) file is generated to capture the simulation waveforms.
- **Firmware Loading**: A specified .hex file is loaded into the testbench memory.
- **Exit Success Check**: The testbench monitors the `exit_success` signal to determine when to end the simulation.
- **Property Checks**: The testbench includes property checks to ensure valid memory accesses and program counter values.

The testbench instantiates three instances of the `system_wrapper` module, each connected to the same RAM and register file. The outputs of these instances are combined using TMR modules to ensure fault tolerance. The testbench also includes instances of the SEC-DED Hamming code encoder and decoder to verify data integrity.

4.2.10 TCL Script for DfT Insertion

A TCL script is used to insert the scan chain, scan compression, programmable LBIST, and test points into the design using Synopsys design compiler. This script automates the process of adding DfT features to the system, ensuring that it can be thoroughly tested.

The script begins by setting up various paths and the top-level module name. It specifies the search paths for libraries and sets the synthetic and target libraries for the design. The script then reads the Verilog files and links the design.

Next, the script defines the DfT signals and creates ports for scan data input, test mode, LBIST enable, scan compression enable, and LBIST test mode. It also creates ports for SEED, SIGNATURE, PATTERN_COUNT, SHIFT_LENGTH, and CAPTURE_CYCLE_ENABLE, and groups them into buses.

The script sets the DfT signals using the created ports and configures the test modes for SCAN, LBIST, and streaming compression. It enables the DfT features, including logic BIST, wrapper, and scan, and sets the testability configuration.

In total, Design Compiler (DC) inserted 1307 test points into the design. The types of test points added were `control_0`, `control_1`, and `observe` (see Figures 4.5, 4.6, and 4.7). The `Control_0` or `control_1` test point forces a signal to a constant 0 or 1 value when `TestMode` is asserted. The `observe` test point is a scan register with its data input connected to the signal to be observed. The insertion of the test point enable logic was done manually by a script in DC. This script added an OR gate with the three enables of the LBIST, scan chain, and scan compression, and then the output drove the enable of the test points.

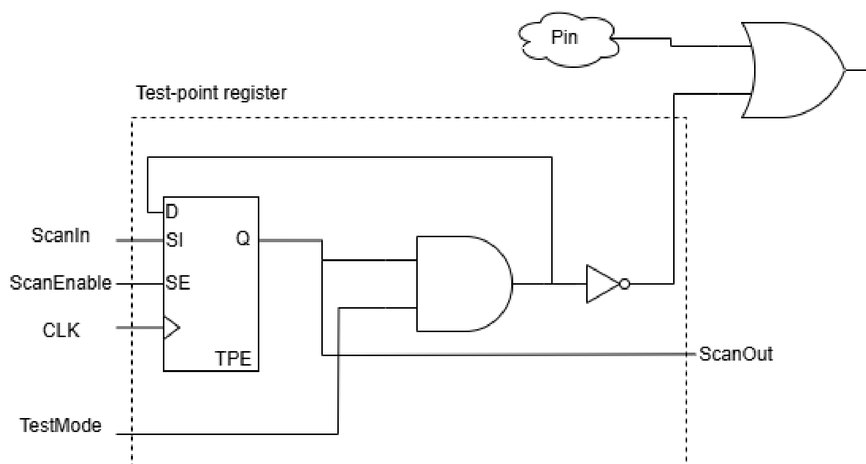

Figure 4.5: Control_0 Test Point from Synopsys DC

The information regarding the test points were taken from these two websites [36, 37].

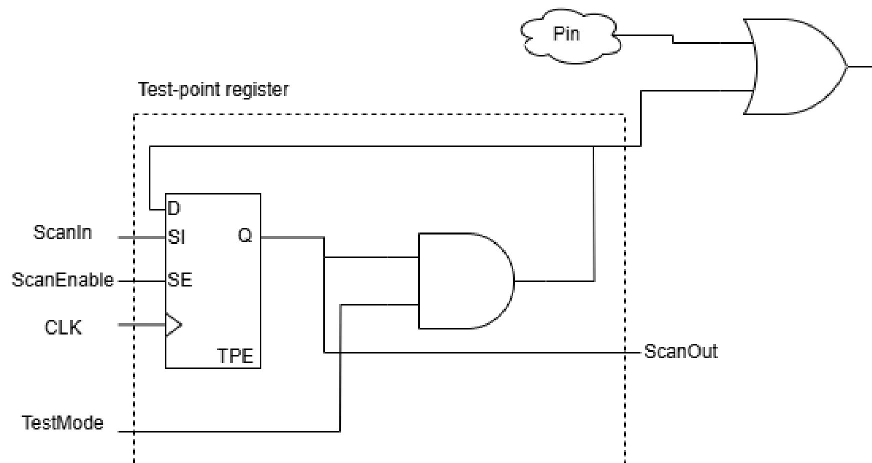

Figure 4.6: Control_1 Test Point from Synopsys DC

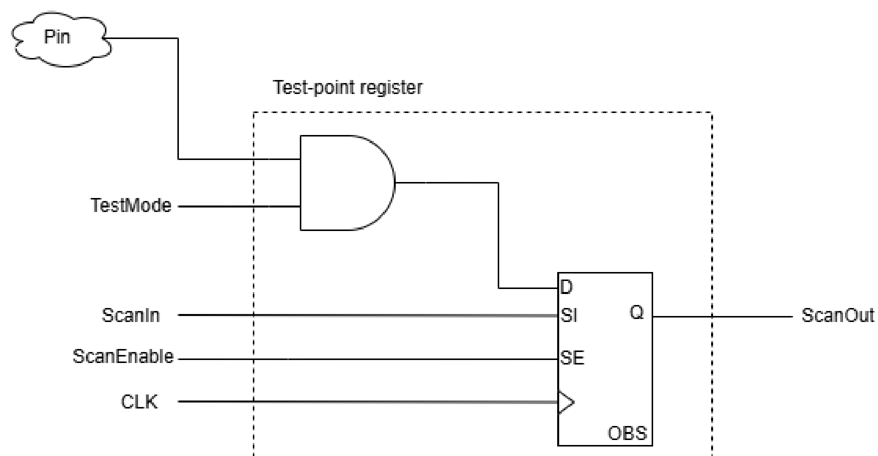

Figure 4.7: Observe Test Point from Synopsys DC

The script then defines the scan paths and includes the necessary segments for LBIST and streaming compression. It previews and inserts the DfT, generates reports, and writes out the design netlist and test protocols for various test modes. Finally, it writes a testbench for Verilog simulation to validate the LogicBIST implementation.

The DfT features inserted by the TCL script are summarized in Table 4.1.

DfT Feature	Description	Purpose
Scan Chain	Inserts scan chains into the design	Enables testing of internal nodes
Scan Compression	Adds scan compression logic	Reduces test time
Test Points	Inserts 1307 test points into the design	Improves test coverage
LBIST	Adds Logic Built-In Self-Test	Enables self-testing of the design

Table 4.1: DfT Features Inserted by TCL Script

Synopsys DfTMAX Compression

Synopsys® DfTMAX™ Compression is a leading scan compression technology designed to make the testing process for ICs more efficient and effective. It is compatible with Synopsys Design Compiler as well as other EDA tools, providing a strong solution for scan compression.

Key Features The key features of Synopsys DfTMAX Compression include ease of integration, multiple test modes, adaptive compression, partitioned compression, power reduction, and high X-tolerance. DfTMAX compression can be added to a design with a single command, making it easy to integrate into existing design flows. The tool supports both standard scan mode and compressed scan mode, offering flexibility in test execution. Adaptive compression allows automatic adjustment of the compression ratio based on the circuit's characteristics, optimizing the testing process and improving test coverage. For larger designs, DfTMAX supports partitioned compression, where multiple codecs are inserted at different levels to manage routing congestion and timing issues. Features such as X-filling and shift power groups help reduce power consumption during testing by minimizing the number of transitions and overall power usage. The tool offers high X-tolerance scan compression, which imposes additional limits on the maximum number of compressed scan chains to ensure 100% X-tolerance.

Compression Architecture The DfTMAX compression architecture includes a codec that consists of a decompressor and a compressor. The decompressor distributes scan input values across many shorter scan chains, while the compressor reduces the captured data from these chains to be observed through the scan-out ports. This architecture significantly reduces test data and test time with minimal silicon area overhead.

Decompressor Operation The decompressor outputs, instead, are driven by different combinations of scan-in data pins, either directly or through multiplexers

(MUXes). This allows the decompressor to adapt to the needs of ATPG and supply the required values in the scan chain cells.

Compressor Operation The compressor outputs are driven by different combinations of compressed scan chains, combined using XOR logic: this creates specific signatures of incorrect values at the compressor outputs when a fault is detected, helping to diagnose the design.

Requirements for Compressed Scan Insertion The requirements for implementing compressed scan insertion include ensuring that the `test_default_strobe` variable is set so that the strobe occurs before the active edges of the test clock waveforms.

To recap, by using Synopsys DfTMAX compression, designers can achieve significant improvements in test efficiency, test coverage, and power consumption, making it an essential tool for modern IC design and testing.

The information provided in this section is derived from [4].

DfTMAX LogicBIST

DfTMAX LogicBIST is a synthesis-based solution designed for the in-system self-test of digital integrated circuits. It is predominantly used in automotive, medical, and aerospace applications to meet functional safety requirements specified by standards such as ISO 26262 for the automotive semiconductor industry.

Key features of LogicBIST include low BIST controller area overhead, reuse of existing scan chain and test-mode control logic, minimal self-test pin requirements, and straightforward interfacing with functional logic. Seed and expected signature values can be either hard-coded or programmable, targeting stuck-at and transition-delay faults. The implementation follows a streamlined one-pass DfT insertion flow.

To implement LogicBIST, several prerequisites must be met: Design Compiler and TestMAX ATPG tools must be installed and licensed, along with DfTMAX or TestMAX DfT tools and an HDL Compiler license for compressed scan insertion. Additionally, blocks must be X-clean, and the integration of self-test logic must be achieved through signal connections to functional logic or DfT-inserted IEEE 1500 logic.

The standard flow for implementing LogicBIST involves several steps. First, LogicBIST DfT logic is inserted into the design. Then, TestMAX ATPG is used to create self-test patterns and compute seed and expected signature values. An autonomous self-test testbench file is written, and the computed seed, signature, and pattern count values are applied. Finally, the netlist and testbench are simulated in a Verilog simulator to verify autonomous BIST operation.

Seed, signature, and pattern count values can be driven by constants in the netlist, resulting in low area overhead but requiring netlist modifications. Alternatively, programmable values can be used, allowing for testing with multiple seed and signature pairs, dividing self-test into small segments, and avoiding constant-value netlist modifications. This is implemented using DfT-inserted IEEE 1500 logic or by connecting self-test signals to internal functional registers.

LogicBIST known issues

Despite its advantages, LogicBIST has known issues. Settings are not stored in .ddc files, designs that capture X values are not supported, and clock-gating cells require a dedicated ScanEnable signal. External chains are not usable in LogicBIST test modes, and multiple LogicBIST test modes are not supported. Unsupported features include pipelined scan enable, DfT partitions, terminal lock-up latches, and hybrid flow.

Note that the information provided in this section is derived again from [4].

Programmable LBIST

Programmable LBIST enhances the flexibility and effectiveness of the traditional LBIST technique by allowing it to drive and monitor internal signals. This programmability enables the customization of test patterns and the ability to target specific areas of the circuit, improving test coverage and reducing test time.

By default, the tool implements the LogicBIST logic with placeholder buses in the netlist for values such as user seed, user signature, user pattern, and user shift values, which are initially tied to logic 0 or set to default values. Instead of setting these to hardcoded constant values in the netlist, it is possible to make them programmable by driving them from ports or internal hookup pins. This is done by defining DfT signals using specific signal types, such as `lbistSeedValue`, `lbistSignatureValue`, `lbistPatternCount`, `lbistShiftLength`, and `lbistCaptureCycleEnable`.

For each signal type, the signal bits are defined in order of most-significant bit (MSB) to least-significant bit (LSB). If fewer signals are defined than the bus width, the signals are justified against the LSB. When internally driven LogicBIST configuration signals are defined, the tool reports the connections on a bitwise basis, allowing confirmation of their correctness.

Additionally, the tool provides procedures to find optimal seed values and to set seed and signature values for simulation. This programmability offers significant advantages by allowing dynamic configuration of test parameters, leading to more efficient and targeted testing processes, ultimately enhancing test coverage and reducing test time.

Once more, the information provided in this section is derived from [4].

4.2.11 TestMAX ATPG Script

The TestMAX ATPG script is used for generating the seed, signature, and number of patterns to achieve determined test coverage. This script configures the ATPG tool to generate test patterns that maximize test coverage and ensure the reliability of the system.

The script begins by setting up various paths, including the root path, gate path, log path, DfT path, and ATPG path. It specifies the search paths for libraries and sets the synthetic and target libraries for the design. Then, it reads the netlist and Verilog files and links the design.

Next, the script builds the ATPG model for the design and writes out the design image. It enables DfTMAX LogicBIST DRC in the TestMAX ATPG flow and sets up the DRC to understand the reset control logic. The script sets up and runs the DRC, adds faults, and runs ATPG with auto compression and JTAG LBIST. It also runs the simulation to validate the generated test patterns.

Finally, it writes the patterns in STIL format and generates a testbench for Verilog simulation to validate the LogicBIST implementation.

4.3 Implementation and Verification Details

4.3.1 Design Verification

Design verification ensures that the implemented design meets the specified requirements and functions correctly. This involves developing a comprehensive verification plan that outlines the test scenarios and coverage goals, running simulations to validate the design functionality and performance under various conditions, and using formal verification techniques to mathematically prove the correctness of critical design components. The verification plan includes defining test cases and establishing coverage metrics, while simulation involves testbench development, inputs generation, and results analysis.

4.3.2 Simulation Tools

Simulation software are essential for verifying the design before physical implementation. Using them involves several steps: choosing appropriate tools based on the design's complexity and verification requirements, setting up the simulation environment including testbenches, inputs, and monitoring mechanisms, and analyzing the simulation results to identify and debug issues in the design. Common tools include Verilog/**Very High-Speed Integrated Circuit Hardware Description Language (VHDL)** simulators and SystemC simulators.

For this project, two widely recognized simulation tools were used: ModelSim and Xcelium.

ModelSim: ModelSim, developed by Mentor Graphics, is a popular HDL simulator for both Verilog and VHDL designs. It provides a comprehensive environment for simulating, debugging, and verifying digital circuits. ModelSim supports mixed-language simulation, making it suitable for complex designs that use both Verilog and VHDL. Moreover its features include advanced waveform viewing, code coverage analysis, and support for various verification methodologies such as UVM (Universal Verification Methodology).

Xcelium: Xcelium, developed by Cadence, is another powerful simulation tool used for verifying digital designs. It supports a wide range of HDLs including Verilog, VHDL, and SystemVerilog. Xcelium is mostly known for its high performance and scalability, making it suitable for large and complex designs. It offers advanced debugging features, coverage analysis, and supports various verification methodologies. The program also integrates well with other Cadence tools, providing a seamless verification flow.

Both ModelSim and Xcelium were used in this project to ensure comprehensive verification of the design. The choice of these tools was based on their robust features and industry-wide acceptance, ensuring that the design meets the required standards and performs as expected.

4.3.3 Synthesis Tools

Synthesis tools translate the Rtl design into a gate-level representation suitable for physical design. Synthesize a design involves choosing synthesis tools that support the target technology and meet performance requirements, defining constraints such as timing, area, and power to guide the synthesis process, and applying optimization techniques to improve the synthesized design performance and efficiency. Common tools include Design Compiler, Genus, and Quartus (for FPGA).

For this project, Design Compiler was used to synthesize the DfT of the SoC. Design Compiler, developed by Synopsys, is a widely used synthesis tool. It provides powerful optimization techniques to meet timing, area, and power constraints, which help the synthesized design to meet the specified requirements.

4.3.4 TestMAX DfT Tool

The TestMAX DfT tool is used for inserting DfT features into the design. This includes adding scan chains, scan compression, LBIST, and test points. The tool automates the process of DfT insertion, and ensures that the design can be thoroughly tested.

The software, developed by Synopsys, provides comprehensive support for various DfT methodologies. As per xcelium it integrates seamlessly with other tools in the design flow, enabling efficient insertion and verification of DfT features. The tool supports advanced DfT techniques, including scan-based testing, test compression, and built-in self-test, which ensure high test coverage and test efficiency.

4.3.5 TestMAX ATPG Tool

The TestMAX ATPG tool is used for generating the seed, signature, and number of patterns to achieve determined test coverage. This tool configures the ATPG process to generate test patterns that maximize test coverage and ensure the reliability of the system.

TestMAX ATPG, also developed by Synopsys, is a comprehensive tool for DfT and ATPG. It supports various test methodologies, including scan-based testing and LBIST. The tool provides advanced features for fault simulation, test pattern generation, and test compression, allowing for a high test coverage with minimal test data volume.

The test parameters generated by the TestMAX ATPG tool are summarized in Table 4.2.

Parameter	Value
Pattern Counter Value	00000000000000001011
PRPG Seed Value	10101100111110010110110001010101001010110110101001101100101010111101000000100100000101101001011 0111
MISR Signature Value	0001110110101111000000000010011
Shift Counter Value	1110100111

Table 4.2: Test Parameters Generated by TestMAX ATPG Tool

4.3.6 Challenges and Solutions

Interface Compatibility

Ensuring compatibility between different interfaces can be challenging. For this project, a protocol mismatch was found between the OBI and the register file. To address this issue, an OBI wrapper was implemented to translate from the OBI protocol to the one used by the register file. This solution ensured seamless communication between the components and maintained data integrity.

General strategies to ensure interface compatibility include adopting industry-standard interfaces (to leverage compatibility and interoperability), implementing bridging logic (to translate between different interface protocols), and conducting thorough testing (to identify and resolve compatibility issues). Standardization and testing, including compatibility and compliance testing, are critical for ensuring seamless communication between interfaces.

Managing Multiple Peripherals

In systems with multiple peripherals, arbitration is essential to manage bus access effectively. Without proper arbitration, simultaneous access attempts by multiple peripherals can lead to data collisions, where signals from different devices interfere with each other, causing data corruption and system instability. Arbitration ensures fair access to the bus, preventing any single device from monopolizing the bus and starving other peripherals of the opportunity to communicate.

For this project, some issues with performance degradation and data corruption were found due to simultaneous access attempts by multiple peripherals. To address this, we utilized efficient arbitration mechanisms and bus segmentation were utilized to manage access and maintain performance integrity.

Efficient arbitration mechanisms optimize overall system performance by minimizing wait times and ensuring that high-priority tasks are handled promptly: this is particularly important in real-time systems where timely data transfer is critical. By managing the order and timing of access requests, arbitration mechanisms reduce latency, improving the responsiveness of the system.

Arbitration also allows for the implementation of priority schemes, where certain peripherals or data transfers are given higher priority over others: this ensures that critical operations, such as real-time data processing or emergency signals, are not delayed by less critical tasks.

4.4 Hardening of the System

Hardening the system involves implementing techniques to enhance its reliability and fault tolerance. This section will cover the key techniques used in this project: TMR and ECC.

4.4.1 Triple Modular Redundancy

TMR is a fault-tolerance technique that involves triplicating critical components and using majority voting to determine the correct output. This method ensures that the system can continue to operate correctly even in the presence of faults.

In this project, TMR was applied to the system wrapper. By triplicating the system wrapper and using majority voting, it is guaranteed that, even if one instance fails, the correct output can still be determined by the other two instances. This significantly enhances the reliability and fault tolerance of the system.

4.4.2 Error Correction Code

ECC is used to detect and correct errors in data storage and transmission by ensuring that errors can be identified and corrected before they affect the system operation.

In this project, ECC was implemented for the memory. Using ECC allows us to detect and correct errors in memory without the need for triplicating the memory area, which would have resulted in an excessively large design. ECC provides a robust mechanism to ensure data integrity with a slight increase in area due to the additional ECC logic. The decision to use ECC for the memory was based on the need to balance fault tolerance with design area constraints, ensuring efficient use of resources while maintaining data integrity. The specific ECC algorithm employed was the Hamming Code SEC-DED, which enables the correction of single-bit errors and detection of double-bit errors.

4.4.3 Implementation Decisions

As said multiple times, hardening the system involves implementing techniques such as TMR and ECC. In this project, is was chosen to: - Triple the Ibex system using TMR to ensure that even if one instance fails, the correct output can still be determined by the other two instances. This approach was chosen for its high reliability in critical processing components. - Implement ECC for the memory to correct single-bit errors and detect double-bit errors. This approach was chosen to avoid the excessive area increase that would result from triplicating the memory, while still maintaining data integrity.

These techniques improve the system reliability, fault tolerance, and data integrity, ensuring that it can operate correctly even in the presence of faults. By incorporating these hardening techniques, the system becomes more robust and resilient, capable of maintaining functionality and data integrity under various fault conditions.

4.5 Memory Selection and integration

Selecting appropriate memory components and implementing MBIST are crucial for ensuring reliable data storage. This involves defining criteria for selecting memory components based on performance, power, and durability requirements, designing the MBIST architecture to automate memory testing and guarantee a high test coverage, and integrating the selected memory components and MBIST into the SoC design. Memory selection criteria include performance metrics, power consumption, and durability, while MBIST architecture includes test pattern generation, response analysis, and control logic.

4.5.1 Memory Selection

Memory selection is a critical aspect of the system design: it involves evaluating different memory technologies such as SRAM, DRAM, and Flash based on their performance, power consumption, and durability characteristics. The selected memory must meet the system requirements for speed, capacity, and durability. Additionally, considerations for memory hierarchy and cache design are essential to optimize performance. One important aspect was ensuring that the module should be available in the technology provided. The table 4.3 provides a comparison of different memory technologies.

Memory Type	Performance	Power Consumption	Storage Type	Cost
SRAM	High	Moderate	Volatile	High
DRAM	Moderate	High	Volatile	Moderate
Flash	Low	Low	Non-volatile	Low

Table 4.3: Comparison of Different Memory Technologies

4.5.2 SRAM integration

A 64KB SRAM was integrated into the design of the CVA6 processor and subsequently synthesized. The SRAM was generated using the STMicroelectronics compiler and manually inserted into the design. This integration is responsible for the memory capabilities of the CVA6 processor. In this configuration of the SRAM, several pins are available to interconnect the MBIST for testing the memory itself. Additionally, there is another available memory for the Ibex. This additional memory resource can be utilized to use the Ibex core, for memory allocation and access for various applications, including running the code for the LBIST.

Chapter 5

Results

This chapter presents the results of the study, focusing on how effective the techniques implemented DfT have been. The system can perform all tests for the included DfT techniques and runs smoothly in normal mode. This means that it can detect faults comprehensively, which increases the overall reliability and efficiency of the SoC. The following sections include detailed tables and graphs along with comments on test coverage, test time, design overhead, and experimental results.

The test coverage presented in this chapter was calculated using the following formula:

$$\text{Test Coverage} = \frac{\text{Detected Faults}}{\text{Total Faults} - \text{Undetectable Faults}}$$

where the undetectable faults are the faults that cannot be observed and controlled due to the netlist design.

5.1 Test Coverage and Test Time

As illustrated in Figure 5.1, the stuck-at faults (SAFs) test coverage percentages for the three test types are as follows:

- **LBIST**: 70.24% with 150 patterns.
- **Scan Chain**: 70.16% with 17 patterns.
- **Scan Compression**: 70.17% with 63 patterns.

The test coverage percentages for all three test types are intentionally made similar to emphasize the comparison of test times. This ensures that the analysis focuses

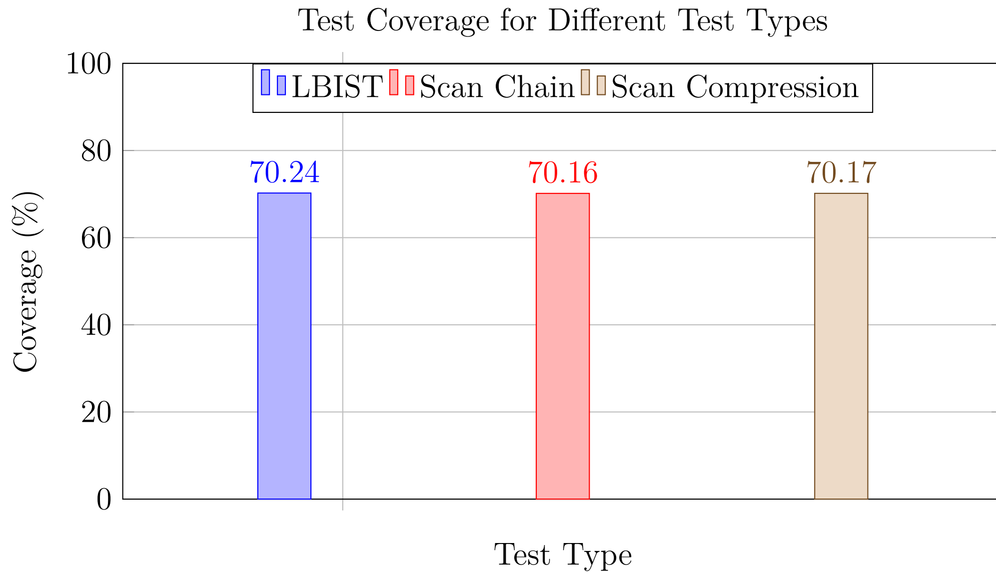

Figure 5.1: Comparison of Test Coverage

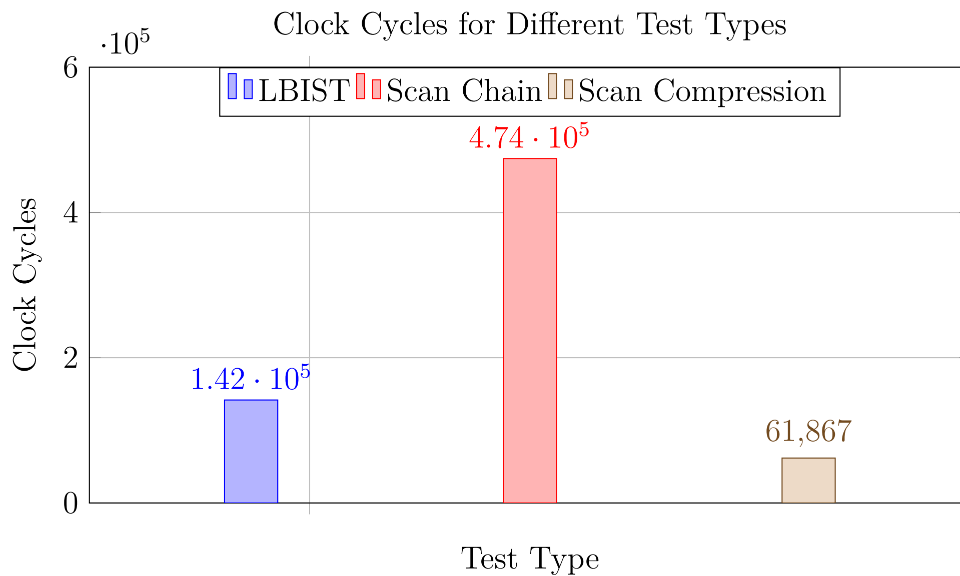

Figure 5.2: Comparison of Clock Cycles

on the efficiency of the test methods in terms of clock cycles, rather than differences in test coverage.

Figure 5.2 displays the number of clock cycles required for each type of test:

- **LBIST**: 141,794 clock cycles.

- **Scan Chain**: 474,283 clock cycles.
- **Scan Compression**: 61,867 clock cycles.

The data reveals that Scan Chain is the most time-consuming method, requiring a significantly higher number of clock cycles compared to the other two methods. In contrast, Scan Compression is the most efficient, requiring the least number of clock cycles. LBIST falls in between, with a moderate number of clock cycles.

5.2 Test Coverage Comparison with and without Test Points

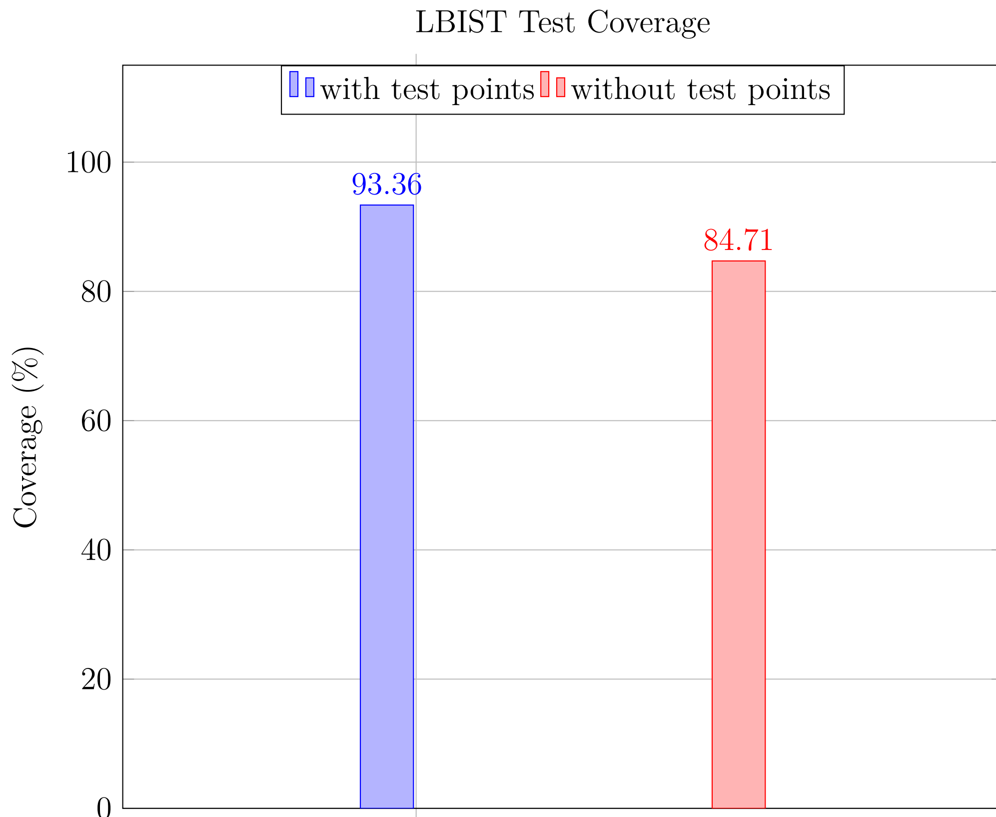

Figure 5.3: LBIST Test Coverage with and without test points

As shown in Figure 5.3, the test coverage of stuck-at-faults with the LBIST with test points is significantly higher at 93.36% compared to 84.71% without test points. The number of patterns used is 600,001 for both cases. This indicates

that the inclusion of test points greatly enhances the fault detection capability of LBIST.

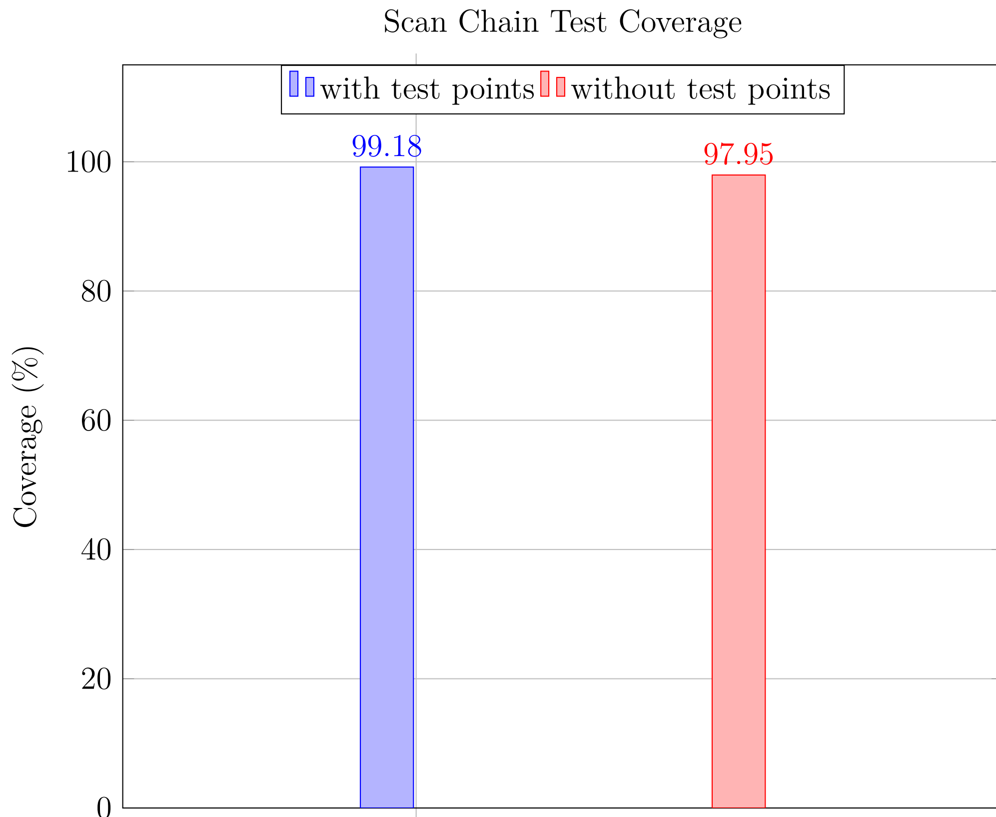

Figure 5.4: Scan Chain Test Coverage with and without test points

Figure 5.4 shows that the test coverage of the scan chain with test points is 99.18%, while without test points, it is 97.95%. The number of patterns used is 1,968 for both cases. Although the improvement is less pronounced than in LBIST, the addition of test points still provides a notable increase in test coverage.

According to Figure 5.5, the test coverage of the scan compression with test points is 99.40%, whereas without test points, it is 97.39%. The number of patterns used is 2,875 for both cases. Similar to the Scan Chain, the inclusion of test points results in a higher test coverage percentage, demonstrating the effectiveness of test points in improving fault detection.

Figure 5.6 shows the test coverage percentages for LBIST, Scan Chain, and Scan Compression with test points. The test coverage values are 93.36%, 99.18%, and 99.40% respectively. It is important to note that for the scan chain and scan compression values, the ATPG was not given a specific number of patterns or test coverage threshold to stop at. Therefore, the test coverage achieved by ATPG

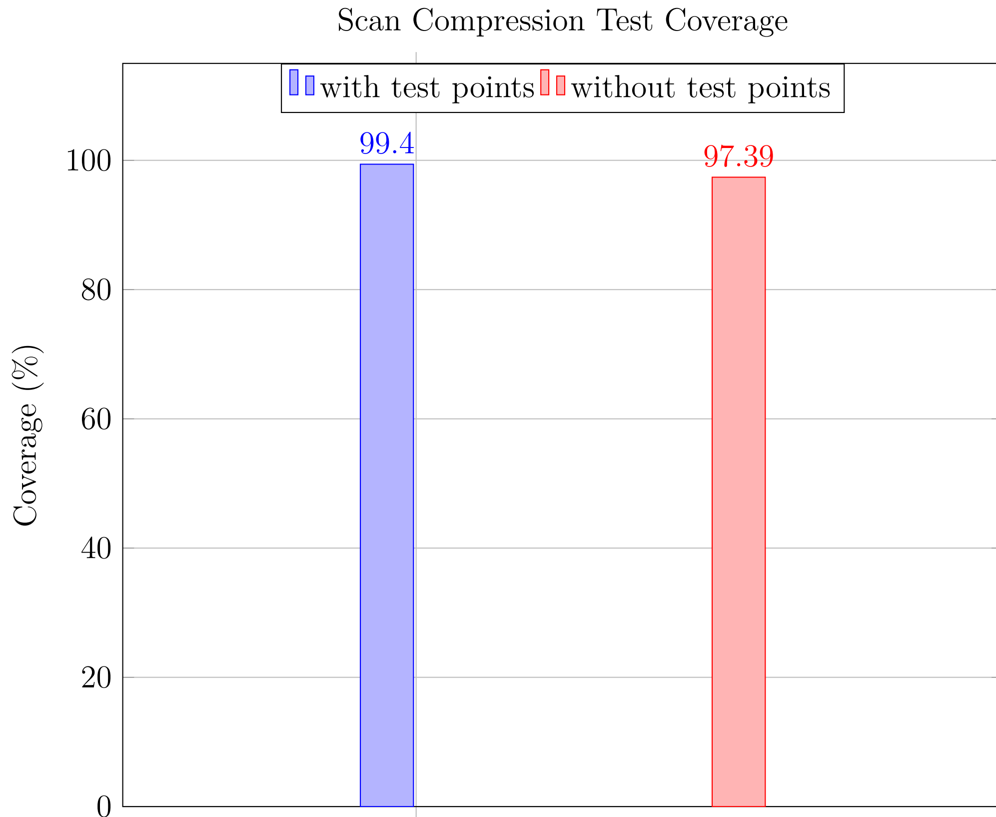

Figure 5.5: Scan Compression Test Coverage with and without test points

before it ceased operation highlights the effectiveness of test points in achieving high test coverage for all three test types. Additionally, the test times for these test types were as follows:

- **LBIST**: 567,176,945 clock cycles.
- **Scan Chain**: 54,905,232 clock cycles.
- **Scan Compression**: 2,823,295 clock cycles.

5.3 Design Area Comparison

The Comparison of the design area before and after DfT insertion (shown in Figure 5.7) shows the impact of DfT techniques on the overall design area of the SoC. The total cell area increased from 1,000,818 μm^2 to 1,024,884 μm^2 after DfT insertion, which is an increase of 2.40%. It is important to note that the area without test

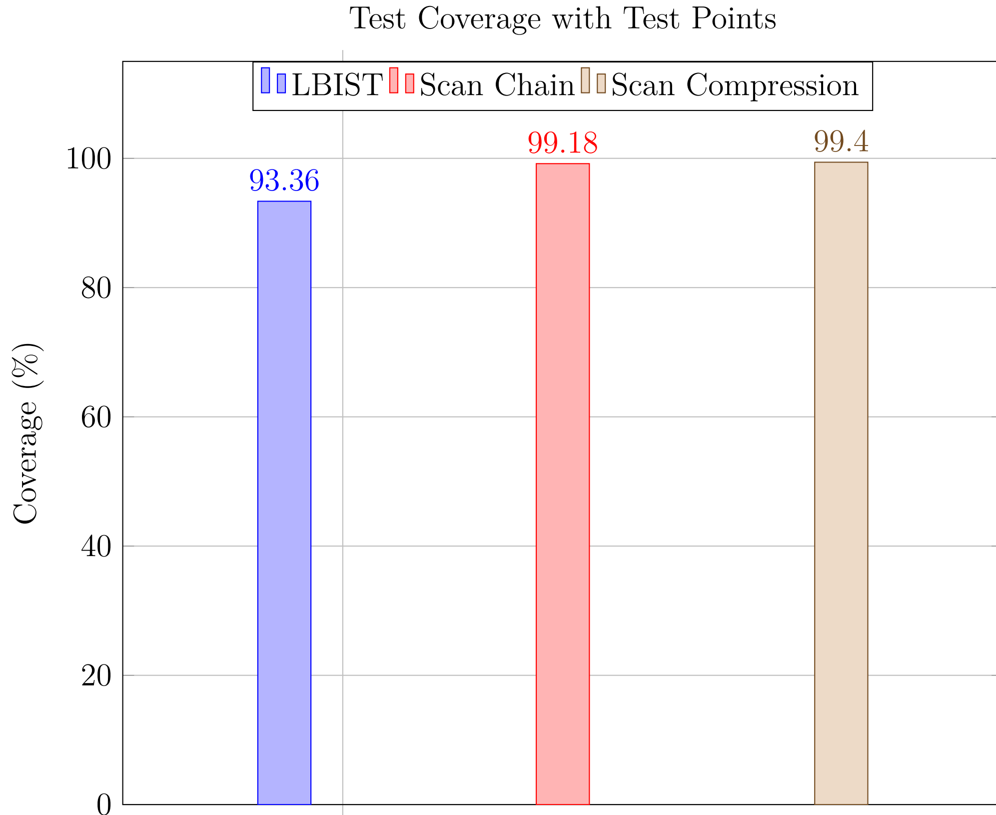

Figure 5.6: Test Coverage for LBIST, Scan Chain, and Scan Compression with Test Points

points is 1,024,108 µm^2. This represents a modest increase of approximately 2.33%. The additional area is mainly due to the inclusion of scan chains, LBIST, and scan compression logic, which are essential for effective testing and fault detection.

5.4 Power Consumption Comparison

The comparison of power consumption before and after DfT insertion reported in Figure 5.8 shows the impact of DfT techniques on the overall power consumption of the SoC. The total power consumption considering internal power, switching power and leakage power has increased from 55.2031 mW to 61.7192 mW after DfT insertion. This increase is due to the additional circuitry required for the DfT techniques. However, we have optimized the power consumption to ensure it remains as low as possible.

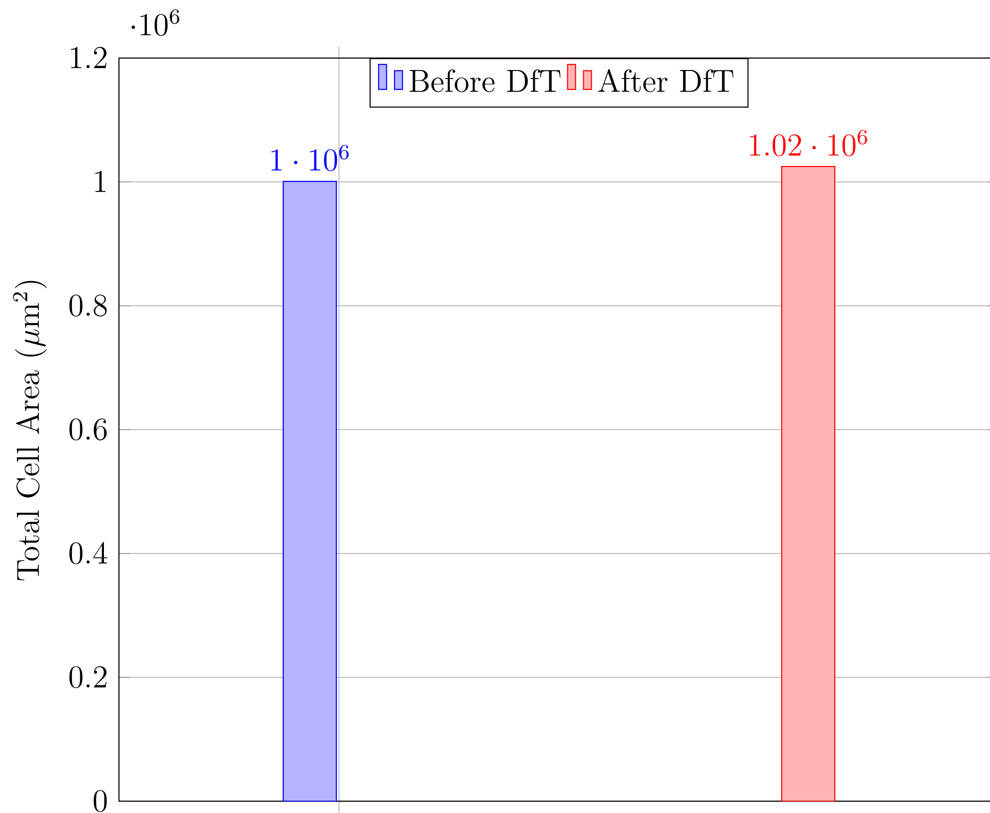

Figure 5.7: Total Cell Area Before and After DfT Insertion (μm^2)

5.5 System Hardening

Regarding the hardening of the system, the Ibex system area is approximately tripled due to the implementation of TMR, as expected. For the memory, instead of triplicating the area, ECC has been used. As a consequence, this has resulted in a slight increase in the memory area due to the additional ECC logic, but it is a more efficient solution compared to triplication.

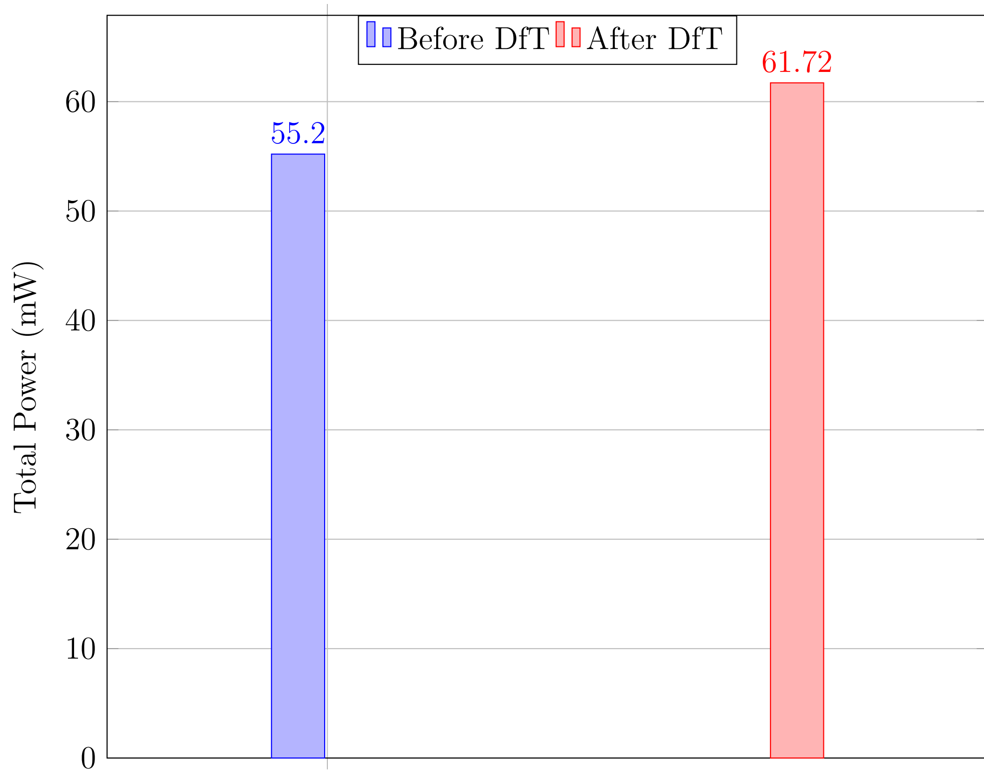

Figure 5.8: Total Power Consumption Before and After DfT Insertion

Chapter 6

Conclusions

6.1 Overview of the Achievements

This chapter provides a comprehensive summary of the work done in this thesis, highlighting the key techniques and methodologies implemented to enhance the reliability, testability, and performance of the System-on-Chip (SoC) design.

6.1.1 DfT Techniques

Design for Testability techniques were extensively explored and implemented to improve the testability of the SoC. The key DfT techniques include:

- **LBIST:** LBIST was implemented to enable automatic testing of logic circuits within the SoC. By embedding test pattern generation and response analysis within the chip, LBIST allows for thorough testing without the need for external test equipment.
- **Scan Chains:** Scan chains were integrated to facilitate the testing of the SoC by converting flip-flops into a series of shift registers. This makes it easier to control and observe the internal states of the circuit during testing, improving test coverage and simplifying the test pattern generation.
- **Test Points:** Test points were strategically placed within the circuit to provide access to internal nodes. This improves test coverage and observability, making it easier to detect and diagnose faults. Test points also facilitate diagnosis by allowing engineers to monitor internal signals and control specific parts of the circuit during testing, identifying and resolving issues more efficiently.
- **Scan Compression:** Scan compression techniques were employed to reduce the volume of test data and the time required for testing. By compressing the

test data before it is applied to the scan chains, it is possible to significantly improve the efficiency of the testing process. This not only reduces the amount of test data that needs to be stored and transferred but also shortens the overall test time, leading to cost savings and faster time-to-market.

- **Programmable LBIST:** Programmable LBIST was implemented to enhance the flexibility and effectiveness of the traditional LBIST technique. By allowing the LBIST to drive and monitor internal signals, it is possible to customize test patterns and target specific areas of the circuit. Programmable LBIST also allows for dynamic reconfiguration of test parameters, making it a versatile and powerful tool for ensuring the reliability of complex SoC designs.
- **Hardening Techniques:** Hardening techniques were employed to enhance the reliability and fault tolerance of the SoC. The key techniques include:
  - **TMR:** TMR was implemented to triplicate critical components and use majority voting to determine the correct output. This ensures that the system can continue to operate correctly even in the presence of faults. By providing redundancy, TMR enhances the fault tolerance and reliability of the SoC, making it suitable for safety-critical applications.
  - **ECC:** ECC was used to detect and correct errors in data storage and transmission. By adding redundancy to the data, ECC allows the system to identify and correct single-bit errors and detect double-bit errors. This boosts data integrity and fault tolerance, ensuring that the SoC can operate reliably even in the presence of data corruption.
- **SRAM:** A 64KB SRAM was integrated into the CVA6 processor, featuring a six-transistor memory cell, high density, low stand-by power, and low dynamic power. The SRAM supports split supply voltages and includes a retention mode for reduced leakage. Additionally, there is another available memory in the Ibex SRAM, enhancing memory allocation and access for various applications.

6.2 Critical Commentary

The implementation of advanced DfT techniques, such as scan compression, programmable LBIST, MBIST, and hardening techniques, has significantly improved the testability, reliability, and performance of the SoC. The results demonstrate that these methodologies are effective in detecting and diagnosing faults, reducing test time, and ensuring data integrity. However, the integration of these techniques also introduced some challenges, such as increased design complexity and area overhead.

Balancing the benefits of enhanced testability and reliability with the associated overheads is crucial for achieving an optimal SoC design. The increased design complexity can lead to longer design cycles and higher development costs. Additionally, the area overhead introduced by the DfT and hardening techniques can negatively impact the overall performance and power consumption of the SoC, other than its cost. Therefore, careful consideration and optimization are required to ensure that the benefits outweigh the costs.

Despite these challenges, the implementation of these techniques has proven to be highly beneficial. The enhanced test coverage and reduced test time contribute to improved product quality and reliability. The ability to detect and correct faults early in the design cycle reduces the risk of costly field failures and product recalls. The integration of TMR and ECC has also proven to be highly effective in enhancing the fault tolerance and reliability of the SoC. These techniques ensured that the system was able to continue to operate correctly even in the presence of faults, making the SoC suitable for safety-critical applications.

The test coverage percentages for LBIST, Scan Chain, and Scan Compression are comparable, ensuring a fair comparison of test times. Scan Compression emerges as the most efficient method in terms of clock cycles, while LBIST is the most time-consuming. It is important to note that both Scan Chain and Scan Compression require an ATE to be tested. In contrast, LBIST is an on-site test that can be performed by any user of the SoC,which generally has a lower test coverage. The inclusion of test points significantly enhances test coverage for all test types, with LBIST benefiting the most from their inclusion. The DfT insertion results in a modest increase in the total cell area, which is a reasonable trade-off for the substantial benefits in test coverage and test time reduction. The power consumption increases after DfT insertion, which is expected due to the additional circuitry. However, the increase is justified by the improvements in test coverage and test efficiency, making the trade-off acceptable. The implementation of TMR for the Ibex system and ECC for memory demonstrates effective strategies for system hardening. While TMR significantly increases the area, ECC provides a more efficient solution for memory protection.

Overall, the work done in this thesis demonstrates the value of incorporating advanced DfT and hardening techniques into modern SoC designs, providing a robust and reliable solution for complex digital systems.

6.3 Future Work

Although this thesis has made significant progress in enhancing the SoC reliability and testability, there are still several features that need to be implemented. The following sections outline the specific areas that require further development to

complete the SoC design:

6.3.1 Interrupt-Based Programmable LBIST

Future work could explore the implementation of interrupt-based programmable LBIST. This would involve designing LBIST mechanisms that can be triggered by interrupts, allowing for more dynamic and flexible testing of the SoC. Such an approach could further reduce test time and improve test coverage by enabling on-demand testing of specific circuit areas. Interrupt-based programmable LBIST could also enhance the ability to perform in-field testing and diagnostics, providing valuable insights into the health and performance of the SoC during operation.

6.3.2 Integration of MBIST with System

The integration of MBIST with the rest of the system to make it programmable by interrupt is a promising area for future research. This integration would involve designing an interface that allows the MBIST controller to communicate with the system interrupt controller. By doing so, memory tests can be dynamically initiated based on specific events or conditions, such as detected memory faults or periodic maintenance checks. This approach would enhance the flexibility and responsiveness of memory testing, ensuring that the system can adapt to changing conditions and maintain high reliability. The MBIST should be generated with the STMicroelectronics compiler and then manually inserted in the design where the pins for the SRAM are available and then connected to the system designed to program it with the Ibex core.

6.3.3 Implementation of the Logic to Handle Double Error Detection

The Double Error Detection signal for the memories and the transmission is only sent to the `obi_wrapper` and the `ibex`, but it is not currently utilized. Implementing the logic to handle this logic should be considered for future improvements.

6.3.4 JTAG

The integration of JTAG standards into the SoC design could be another area of future research. JTAG provides a standardized interface for testing and debugging integrated circuits, facilitating the testing of complex SoC designs. Implementing JTAG could enhance the accessibility and control of internal nodes, further improving the testability and reliability of the SoC. Additionally, JTAG could enable advanced debugging and diagnostic capabilities, allowing engineers to identify and

resolve issues more efficiently.

Overall, the work done in this thesis lays a foundation for future research and development in the field of SoC reliability and testability. By continuing to explore and implement advanced testing and hardening techniques, it is possible to develop more reliable, efficient, and robust SoC designs that meet the demands of modern electronic systems. The ongoing advancements in DfT, programmable LBIST, MBIST, and hardening techniques will play a crucial role in shaping the future of SoC design, ensuring that these complex systems can operate reliably and efficiently in a wide, and expanding range of applications.

References

[1] Laung-Terng Wang, Charles E. Stroud, and Nur A. Touba. *System-on-Chip Test Architectures: Nanometer Design for Testability*. Morgan Kaufmann, 2006 (cit. on pp. 2, 14).

[2] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kaufmann, 2006 (cit. on p. 2).

[3] Jose Luis Huertas. *Test and Design-for-Testability in Mixed-Signal Integrated Circuits*. Springer, 2004 (cit. on p. 2).

[4] Synopsys. *DFT User Guide*. https://spdocs.synopsys.com/dow_retrieve/qsc-u/dg/dftolh/U-2022.12/dftolh/pdf/dftug.pdf. 2022 (cit. on pp. 7, 14, 16, 18, 54, 55).

[5] Wikipedia. *Design for testing*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Design_for_testing (cit. on p. 8).

[6] Wikipedia. *Fault model*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Fault_model (cit. on p. 9).

[7] Wikipedia. *Stuck-at fault*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Stuck-at_fault (cit. on p. 10).

[8] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990 (cit. on p. 12).

[9] Paul H. Bardell, William H. McAnney, and Jacob Savir. *Built-In Test for VLSI: Pseudorandom Techniques*. John Wiley & Sons, 1987 (cit. on p. 14).

[10] Digital Electronics Blog. *Designing for Debug: Top 10 Strategies*. https://blog.digitalelectronics.co.in/2023/03/designing-for-debug-top-10-strategies.html?m=1. 2023 (cit. on p. 16).

[11] Matteo Sonza Reorda. *Test Points in Digital Design*. https://file.didattica.polito.it/dl/MATDID/33627591. 2024 (cit. on p. 16).

[12] Ronald D. Schrimpf and Dan M. Fleetwood. *Radiation Effects and Soft Errors in Integrated Circuits and Electronic Devices*. World Scientific Publishing Company, 2004 (cit. on p. 18).

[13] Matteo Sonza Reorda. *Error Correcting Codes in Digital Systems*. `https://file.didattica.polito.it/dl/MATDID/33660868`. 2024 (cit. on p. 18).

[14] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Kluwer Academic Publisher, 2000 (cit. on p. 18).

[15] Alfred Crouch. *Design for Test: For Digital IC's and Embedded Core Systems*. Prentice Hall PTR, 1999 (cit. on p. 18).

[16] Wikipedia. *System on a Chip*. Accessed: 2023-10-10. 2023. URL: `https://en.wikipedia.org/wiki/System_on_a_chip` (cit. on p. 19).

[17] Wayne Wolf. *Modern VLSI Design: System-on-Chip Design*. Prentice Hall, 2008 (cit. on p. 19).

[18] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC, 2017 (cit. on p. 20).

[19] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017 (cit. on p. 20).

[20] Wikipedia. *RISC-V*. Accessed: 2023-10-10. 2023. URL: `https://en.wikipedia.org/wiki/RISC-V` (cit. on p. 20).

[21] PULP Platform. *Ibex Core Documentation*. `https://ibex-core.readthedocs.io/`. 2023 (cit. on p. 25).

[22] OpenHW Group. *OBI Protocol Documentation*. `https://docs.openhwgroup.org/projects/obi-protocol/`. 2023 (cit. on p. 25).

[23] PULP Platform. *CVA6 Core Documentation*. `https://docs.openhwgroup.org/projects/cva6-user-manual/index.html`. 2023 (cit. on p. 26).

[24] Claudio Passerone. *RAM Documentation*. `https://file.didattica.polito.it/download/MATDID/33430010`. 2023 (cit. on p. 29).

[25] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2002 (cit. on p. 29).

[26] Ankit Kumar. *What are the various types of memory faults?* Accessed: 2023-10-10. 2023. URL: `https://www.linkedin.com/pulse/what-various-types-memory-faults-kumar-ankit-tlclf` (cit. on p. 29).

[27] Wikipedia. *JTAG*. Accessed: 2023-10-10. 2023. URL: `https://en.wikipedia.org/wiki/JTAG` (cit. on p. 31).

[28] Wikipedia. *Scan chain*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Scan_chain (cit. on p. 31).

[29] Wikipedia. *Built-in self-test*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Built-in_self-test (cit. on p. 32).

[30] Wikipedia. *Logic built-in self-test*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Logic_built-in_self-test (cit. on p. 33).

[31] Wikipedia. *Triple modular redundancy*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Triple_modular_redundancy (cit. on p. 34).

[32] Wikipedia. *Error correction code*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Error_correction_code (cit. on p. 34).

[33] Wikipedia. *ECC memory*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/ECC_memory (cit. on p. 34).

[34] Wikipedia. *Automotive electronics*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Automotive_electronics (cit. on p. 36).

[35] Wikipedia. *Avionics*. Accessed: 2023-10-10. 2023. URL: https://en.wikipedia.org/wiki/Avionics (cit. on p. 36).

[36] Synopsys, Inc. *Observe Test Points*. Accessed: 2023-10-15. 2022. URL: https://spdocs.synopsys.com/dow_retrieve/qsc-u/dg/dftolh/U-2022.12-SP1/dftolh/dftug/advanced_dft_architecture_methodologies/observe_test_points.html (cit. on p. 51).

[37] Synopsys, Inc. *Control Test Points*. Accessed: 2023-10-15. 2022. URL: https://spdocs.synopsys.com/dow_retrieve/qsc-u/dg/dftolh/U-2022.12-SP1/dftolh/dftug/advanced_dft_architecture_methodologies/control_test_points.html (cit. on p. 51).