MEMORY BUILT IN SELF TEST INSERTION IN SOC FOR TESTING AND VALIDATION

A Thesis Submitted in Partial Fulfilment of the Requirements for

the Degree of

MASTER OF TECHNOLOGY

in

VLSI DESIGN AND EMBEDDED SYSTEMS

by

Sahil Srivastava

(2K24/VLS/25)

Under the Supervision of

Mr. Kaustubh Ranjan Singh

(Assistant Professor) DTU, Delhi

Department of Electronics and Communication Engineering

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad, Daulatpur, Bawana Road, Delhi-110042

May, 2026

Department of Electronics and Communication Engineering
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad, Daulatpur, Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I, Sahil Srivastava, Roll No. 2K24/VLS/25 hereby certify that the work which is being presented in the thesis entitled ***'Memory Built in Self Test Insertion in SOC for Testing and Validation'*** in partial fulfilment of the requirements for the award of the Degree of Master of Technology (VLSI and Embedded Systems), submitted in the Department of Electronics and Communication Engineering, Delhi Technological University, is an authentic record of my own work carried out during the period from January 2026 to May 2026 under the supervision of **Mr. Kaustubh Ranjan Singh**.

I have not submitted the matter presented in the thesis for the award of any other degree of this or any other Institute.

Sahil

Candidate's Signature

Sahil Srivastava (2k24/VLS/25)

Department of Electronics and Communication Engineering
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad, Daulatpur, Bawana Road, Delhi-110042

CERTIFICATE BY THE SUPERVISOR

This is to certify that the work reported in this thesis entitled *"Memory Built In Self Test Insertion in for SOC Testing and Validation"* has been carried out by Mr. Sahil Srivastava (2K24/VLS/25) under my supervision.

Supervisor's Signature
Mr. Kaustubh Ranjan Singh
Asst. Professor ECE, DTU, Delhi

Department of Electronics and Communication Engineering
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering) Shahbad, Daulatpur, Bawana Road, Delhi-110042

PLAGIARISM VERIFICATION

Title of the Thesis: Memory Built In Self Test Insertion in SOC for Testing and Validation

Total Pages: Eighty-Six

Name of the Scholar: **Sahil Srivastava**

Supervisor: **Mr. Kaustubh Ranjan Singh**

Department of Electronics and Communication Engineering

This is to report that the above thesis was scanned for similarity detection. The process and outcome is given below:

Software used: **Turnitin** Similarity Index: 9 % Total Word Count: 17,460

Date: ____________

Candidate's Signature

Signature of Supervisor(s)

ACKNOWLEDGEMENT

The successful completion of any task is incomplete and meaningless without giving any due credit to the people who made it possible without which the project would not have been successful and would have existed in theory. I would like to express my deep sense of gratitude and indebtedness to my highly respected and esteemed faculty in-charge of the minor project for having suggested the topic of my minor project and for giving me complete freedom and flexibility to work on this topic. They have been very encouraging and motivating and the intensity of encouragement has always increased with time. Without their constant support and guidance, I would not have been able to attempt this project. I extend my sincere thanks to all my colleagues who have patiently helped me directly or indirectly in accomplishing this minor project work successfully.

Place: Delhi

Date: 31st May 2026

Name: SAHIL SRIVASTAVA

Roll No.: 2k24/VLS/25

ABSTRACT

As semiconductor technology continues to advance, the integration of complex and high-density memory architectures in modern electronic systems has become ubiquitous. With aggressive scaling, reduced feature sizes, and increased operational frequencies, memory devices are increasingly susceptible to a wide range of defects and faults that can compromise the reliability, performance, and functional safety of the entire system. To address these concerns, Memory Built-In Self-Test (MBIST) and Memory Repair techniques have emerged as indispensable components in ensuring the long-term integrity and robustness of memory subsystems.

MBIST provides an automated, highly efficient mechanism for testing embedded memories during manufacturing, system initialization, and even periodic in-field operation. By embedding test logic directly within the chip, MBIST eliminates the dependency on costly external testers, reduces test time, and enables comprehensive fault coverage through advanced algorithms such as March-based tests, checkerboard patterns, and algorithmic stress sequences. These capabilities allow for early detection, isolation, and diagnosis of memory-related faults, ultimately improving product quality, accelerating validation cycles, and enhancing overall system reliability. The integration of MBIST further contributes to optimizing manufacturing yields by allowing precise fault characterization at the wafer and package levels.

Memory repair techniques complement MBIST by mitigating the impact of permanent or hard-to-correct defects discovered during testing. These methodologies employ redundancy mechanisms—such as spare rows, spare columns, redundant blocks, and error-tolerant architectures—to reconfigure around faulty memory locations. Once defects are identified, repair logic redirects memory accesses to healthy redundant elements, effectively salvaging the memory module. Although redundancy introduces certain trade-offs in terms of silicon area, power, timing complexity, and repair latency, it significantly reduces the number of unusable dies and enhances the overall lifetime reliability of the system.

Emerging architectures such as multi-port SRAMs, embedded DRAMs, non-volatile memories, and FinFET-based designs require adaptable test strategies and scalable repair frameworks. Additionally, as technology nodes shrink, new fault modes—such as data retention failures, leakage-induced faults, and timing-dependent defects—necessitate more sophisticated testing approaches. In safety-critical domains like automotive, aerospace, and medical electronics, achieving high diagnostic coverage, fault tolerance, and compliance with functional safety standards becomes essential. Consequently, MBIST and memory repair systems must continue to evolve to meet the stringent reliability demands of next-generation semiconductor devices.

LIST OF PUBLICATIONS

1. **Kaustubh Ranjan, Sahil Srivastava** “Optimization of MBIST March Algorithms Using Structural Redundancy Pruning and Machine Learning Guided Configuration Selection” .Submitted, Unpublished in the 6th International Conference on Emerging VLSI & Semiconductor Technology for AI and Computing Applications (EVST–2026)

Table of Contents

Title	**I**
Candidate's Declaration	**II**
Certificate by the Supervisor	**III**
Plagiarism Verification	**IV**
Acknowledgment	**V**
Abstract	**VI**
List of Publications	**VIII**
Table of Contents	**IX**
List of Figures	**XI**
List of Tables	**XII**
List of Abbreviations	**XIII**
Chapter 1: Introduction	**1-14**
1.1 DFT	2
1.1.1 MBIST	4
1.1.2 IJTAG	6
1.1.3 Tessent	9
1.1.4 VCS	11
1.1.5 Verdi	12
1.2 Motivation	13
1.3 Problem Definition	13
1.4 Objectives	14
Chapter 2 : Literature Review	**15-17**
Chapter 3 : Methodology	**18-47**
3.1 MBIST Architecture	18
3.2 MBIST Insertion	19
3.3 IEEE 1687 IJTAG Architecture	23
3.4 TAP Creation	29
3.5 Pattern Generation and Verification	30
3.5.1 Creating simulation testbench patterns	31
3.5.2 Creating manufacturing test patterns	33
3.5.3 Generated MBIST verification patterns	35
3.6 Memory BIST Diagnosis	36
3.7 Implementation and Verification of Memory Repair	38
3.8 Algorithms	41
3.9 Methodological Framework	44
Chapter 4 : Results and Analysis	**48-57**

Chapter 5 : Conclusion, Future and Social Impact **58-63**
5.1 Conclusion 58
5.2 Future Scope 59
5.3 Social Impact 61

References **64**
List of Publication and Proof **66**
Plagiarism Report **67**
Curriculum Vitae **73**

List of Figures

3.1 MBIST Architecture 19
3.2 Flow chart of Top-Down insertion flow 21
3.3 Circuit before BIST insertion 21
3.4 Circuit after BIST insertion 22
3.5 IEEE 1687 Architecture 25
3.6 An example embedded instrument showing location of PDL Enables IP portability 26
3.7 Pattern Generation Flow 31
3.8 Standard Diagnostic Approach Using Parallel Compare Status Lines 37
3.9 Diagnostic Approach Using Stop-On-Nth-Error Serial Scan 38
3.10 Chip level repair architecture 39
3.11 Example of a memory with redundant IO 41
3.12 Example of a memory with redundant column 41
4.1 Overall Project Flow diagram 49
4.2 Files generated after MBIST insertion 50
4.3 Additional logic that got added post MBIST insertion 50
4.4 Additional instances that got added post MBIST insertion 51
4.5 Waveform of SRAM ports in Synopsys Verdi 51
4.6 Diagnostic pattern simulation report 52
4.7 Simulation log before optimization of Algorithms 56
4.8 Simulation log after optimization of Algorithms 57

List of Tables

Table 3.1 Step Level Execution Sequence of March C 43

Table 3.2 Step Level Execution Sequence of March LR 43

Table 3.3 Step Level Execution Sequence of March SS 43

Table 3.4 Step Level Execution Sequence of March Y 44

Table 3.5 Step Level Execution Sequence of March SR 44

Table 4.1 Local Step Redundancy of all March Algorithms 55

Table 4.2 Global Step Redundancy of all March Algorithms 55

List of Abbreviations

MBIST: Memory Built in Self Testing

IC: Integrated Chip

DFT: Design for Testability

JTAG: Joint Test Action Group

PCB: Printed Circuit Board

SRAM: Static Random Access Memory

DRAM: Dynamic Random Access Memory

SoC: System on Chip

FSM: Finite State Machine

BIRA: Built in Redundancy Analysis

ICL: Instrument Connection Language

PDL: Procedural Description Language

TAP: Test Access Port

VCS: Virtual Component Solution

Chapter 1

Introduction

Chip testing is a critical phase in the semiconductor manufacturing process, designed to ensure that integrated circuits (ICs) operate correctly and are free from defects. This testing is carried out at various stages, including during design verification, post-manufacturing, and before the chips are packaged and shipped. Chip testing ensures the functionality, performance, and reliability of the semiconductor device. The testing process can be divided into two primary types:
Functional Testing: Verifies whether the chip operates according to its intended design specifications. This testing checks the logical correctness of the IC, ensuring that all functional elements work together as required. It typically involves running test vectors to simulate real world scenarios.
Structural Testing: Focuses on identifying manufacturing defects by checking the physical structure of the chip, ensuring that all components and connections are intact. This includes methods like DFT and Built-In Self-Test (BIST) to detect issues at every node, even if they don't impact functionality directly.

Over the years, DFT techniques have evolved to enhance chip testability and streamline the testing process, especially as chips have become more complex with smaller process nodes and higher integration. Some key advancements in DFT include: Scan Chains: One of the foundation techniques in DFT, scan chains convert sequential elements in the design (like flip-flops) into controllable and observable elements. This allows designers to access internal states of the chip for testing, improving fault detection.

Boundary Scan: Introduced as the IEEE 1149.1 standard (JTAG), boundary scan provides a way to test interconnects on printed circuit boards (PCBs) and ICs without physical access to all pins. This is particularly useful for complex systems where pin access is limited. Built-In Self-Test (BIST): BIST enables chips to test themselves by embedding test logic within the IC. This self-testing capability reduces reliance on external test equipment and allows for faster, on-chip fault detection.

At-Speed Testing, with shrinking process nodes and higher operational frequencies, traditional low-speed testing may not reveal issues that only occur at full operational speed. At-speed testing methods are designed to simulate real-time chip operations, capturing defects that might be missed during slower tests.

Compression Techniques: To cope with the growing size and complexity of chips, modern DFT includes test data compression techniques, which reduce the volume of test data that needs to be applied. This helps minimize test time and cost without sacrificing coverage.
As memory content in chips grows, testing embedded memories like SRAM, DRAM, or flash has become increasingly challenging. MBIST is a specialized type of BIST developed for memory arrays. MBIST offers several benefits:
Efficient Memory Testing: MBIST algorithms can efficiently test large memory arrays for common defects such as stuck-at faults, coupling faults, and bridging faults. By embedding the test logic directly within the chip, MBIST simplifies testing across all memory instances.
Automation: MBIST automates the testing of memory components, reducing the need for external testers and allowing for faster execution. It can be programmed to test various memory faults through patterns like March tests or moving-inversion tests.
Integration with DFT: MBIST can be combined with other DFT techniques to provide comprehensive coverage. For example, MBIST may be run in parallel with scan chain tests, further increasing the efficiency of the testing process.

1.1 DFT

Design for testing or design for testability (DFT) consists of IC Design techniques that add testability features to a hardware product design. The development and

application of manufacturing tests to the designed hardware is facilitated by the additional features. Manufacturing tests are performed to confirm that there are no manufacturing flaws in the product hardware that might compromise its proper operation. DFT is included in the chip design and manufacturing process to guarantee that no manufacturing defects of any type occur in any of the chip's nodes. This entails creating the necessary test inputs and outputs, defining test features for the chip, and once the chip is manufactured, testing the chip for flaws.

For semiconductor design and manufacturing organizations, DFT is crucial because it helps prevent the enormous costs and delays that arise from discovering faults after the chip has been sold to the client. It is a preventative measure to separate out individual packed units and faulty dies. Tests for functional verification are not related to DFT. Functionality tests are necessary to confirm that the chip fulfills its intended purpose, and DFT examines every node regardless of its functional role to guarantee that errors and manufacturing defects that are overlooked in functionality tests are also found and fixed [1]. Key elements of DFT include:

Testability Features: These are design techniques embedded into the IC layout to enable comprehensive testing. Examples include scan chains, built-in self-test (BIST) structures, and test access mechanisms (TAMs) that facilitate the application and observation of test patterns.

Manufacturing Tests: These tests are applied during production to verify the correctness of the fabricated ICs. They aim to detect manufacturing flaws that could impair the functionality or reliability of the final product.

Preventive Measures: DFT serves as a preventive measure against the discovery of faults after the ICs have been deployed in real-world applications. This helps semiconductor companies avoid costly recalls, reputation damage, and customer dissatisfaction.

Complementary with Functional Testing: While functional tests validate whether the chip performs its intended operations correctly, DFT ensures that every node within the chip is thoroughly inspected for defects, even those not directly related to functionality. This comprehensive approach enhances overall product quality.

At a higher level there is a central TAP which is connected to the BIST engines which are scattered across the chip, The TAP is connect to BIST engines through BAP. TAP is also connected the BISR controller to access the repair information. [4]

1.1.1 MBIST

A significant portion of VLSI circuits are made of memories. The design of memory systems aims to store enormous volumes of data. Flip-flops and logic gates are not part of memories. Therefore, in order to test memory, several defect models and test procedures are needed. The self-testing and repair mechanism known as memory built in self test (MBIST) examines the memories using a powerful set of algorithms to identify potential faults such as coupling (CF), neighbourhood pattern sensitive faults (NPSF), stuck-at (SAF), and transition delay faults (TDF) that may exist inside a typical memory cell. To create the test patterns for the test, it makes use of an integrated clock, address and data generators, and read/write controller circuitry.

Since memory cells in a common memory model are interconnected in a two-dimensional array, it is necessary to analyse memory cell performance in relation to the array structure. The "storage node" and the "select device" are the two main parts of the memory cell in the array structure. The memory cell can be addressed for read/write access in an array with the help of the "select device" component. Both of these factors affect memory scaling constraints.

External test patterns are applied as a stimulus during the automated tested equipment (ATE) testing procedure for the produced chip design verification. The tester examines the device's reaction and compares it to the golden response, which is kept in the test pattern data [9]. With advancements in semiconductor technology, the size and complexity of memory in ICs have grown tremendously. Testing these embedded memories using traditional external test equipment becomes challenging due to factors such as:

Access Issues: Many embedded memories are deeply integrated within the chip and are difficult to access externally without disrupting the surrounding logic.

Test Time and Cost: As the memory size grows, applying test patterns from external testers becomes more time-consuming and expensive.

Fault Coverage: Embedded memories are prone to defects like stuck-at faults, transition faults, and coupling faults, which need thorough testing to ensure reliability.

MBIST addresses these challenges by incorporating test logic into the IC design, allowing the memory to test itself with minimal external intervention. This is crucial for achieving high fault coverage with reduced test time and cost. MBIST, which incorporates all of these features into a test circuit that surrounds the memory on the chip. It uses a finite state machine (FSM) to create stimuli and evaluate the responses that come from recollections. The additional self-testing circuitry serves as the memory's and the high-level system's interface. This interface reduces the difficulties associated with testing embedded memories by enabling controllability and observability. The requirement for an external test pattern set for memory testing is significantly reduced because the FSM offers test patterns for memory testing [1]. Key Features of MBIST Include:

Self-Sufficiency: MBIST enables memory to test itself using built-in circuitry. This reduces dependency on external Automatic Test Equipment (ATE), thereby lowering test costs and improving scalability.

Automated Test Execution: The MBIST controller is responsible for generating test patterns, applying them to the memory, and analyzing the results. This process is automated and can be executed at different stages of the chip's lifecycle, including during production and in-field testing.

Fault Diagnosis: In addition to detecting memory faults, MBIST can also provide diagnostic information to help locate the exact memory bit or array causing the failure. This diagnostic capability is especially important for yield improvement during chip manufacturing.

Built-In Repair Mechanism: Some advanced MBIST implementations are integrated with Built- In Redundancy Analysis (BIRA) or Built-In Repair Analysis (BIRA). This allows defective memory cells to be bypassed or replaced with spare rows or columns, improving the yield of the manufacturing process.

Test Algorithms: MBIST can execute various memory test algorithms to detect different types of memory faults. Some commonly used algorithms include:

March Tests: A widely used class of memory test algorithms that detect common faults like stuck-at, transition, and coupling faults.

Walking 1/0 Test: A simple test pattern where a single bit is moved through the memory array to detect address and data line issues.

Checkerboard/Inverse Checkerboard: This pattern alternates 1s and 0s in a checkerboard arrangement, testing for coupling faults between adjacent memory cells.

Moving Inversion Test: A test that writes a pattern to the memory and reads it back after inverting all the bits to check for faults like stuck-at and transition faults.

MBIST has many advantages such as,

Reduced Test Time: Since MBIST operates at the internal clock speed of the IC, it can test large memory arrays faster than traditional external testing methods, which are often limited by the speed of external test equipment.

Cost-Effectiveness: MBIST reduces the need for expensive external testers, especially for large memory arrays. The test can be conducted on-chip, reducing the overall test cost.

Improved Test Coverage: MBIST provides high fault coverage by enabling more comprehensive test algorithms and accessing internal nodes that may be difficult to reach with external testing.

In-Field Testing: MBIST can be used for post-production testing, such as in-field diagnostics or reliability checks, making it ideal for mission-critical applications where long-term reliability is essential.

Support for Memory Repair: In cases where redundant memory cells are available, MBIST can be combined with repair mechanisms to replace faulty memory cells, improving overall yield.

1.1.2. IJTAG

IJTAG, formally known as IEEE 1687-2014, is an extension of the traditional JTAG (Joint Test Action Group) standard, specifically created to address the growing

complexity of system-on- chip (SoC) designs that contain numerous embedded instruments. While the traditional JTAG (IEEE 1149.1) was initially developed for testing and debugging at the printed circuit board (PCB) level, the increased use of embedded test structures within ICs required a more sophisticated mechanism for access and control. IJTAG was introduced to manage these instruments within complex integrated circuits, offering more flexibility and scalability. IJTAG provides a standardized method to access, configure, and control embedded instruments, such as: Built-In Self-Test (BIST) engines, Power and temperature sensors, Performance monitors, Memory BIST (MBIST), Other test, measurement, or debug instruments integrated within an IC. Key features of IJTAG are,

Instrument Connectivity Language (ICL): ICL is used to describe the hardware structure of embedded instruments within an SoC and how they are interconnected. It essentially defines the instrument network topology within the chip. ICL specifies the access protocols, addressing, and the paths required to connect to each instrument. It enables automated procedures for accessing and managing the instruments. By providing a standardized description of the hardware connections, ICL helps streamline the integration of embedded instruments into the test flow, allowing for the automatic generation of access procedures during test development.

Procedural Description Language (PDL): PDL complements ICL by describing the procedures and operations that can be performed on the embedded instruments. This includes details on how to Initialize an instrument, configure it, retrieve data from it, Execute tests or diagnostic routines. PDL is used to automate the execution of these procedures, enabling efficient communication between test systems and embedded instruments. With PDL, designers can create reusable scripts to access instruments without having to know the low-level details of how they are connected. This simplifies test development and increases test automation.

Network Reconfiguration: IJTAG introduces a mechanism for dynamic network reconfiguration, which allows the instrument access network to be reconfigured in real-time. This dynamic reconfiguration allows for selective bypassing of instruments that are not needed for a particular test, reducing access time and optimizing the performance of the test. By dynamically adjusting the network configuration, IJTAG enables efficient testing of specific instruments without the need to traverse the entire network, improving test speed and reducing overall test complexity.

Scalability: One of the strengths of the IJTAG standard is its scalability. It is designed to support a wide range of device sizes, from small ICs with a few embedded instruments to large SoCs with hundreds of instruments. The IJTAG architecture can handle SoCs that contain a diverse set of instruments and hierarchical structures, making it ideal for use in modern semiconductor devices that integrate various sensors, test engines, and performance monitors. IJTAG provides a scalable solution that grows with the complexity of the chip, without significantly increasing overhead.
Integration with Existing JTAG Infrastructure: IJTAG is designed to work seamlessly with the existing JTAG infrastructure (IEEE 1149.1). It leverages the TAP (Test Access Port) interface and does not require any additional pins beyond the standard JTAG four- or five-pin interface. This backward compatibility ensures that IJTAG can be adopted in existing test setups without requiring significant hardware or process changes. It makes it easier for semiconductor companies to integrate IJTAG into their current workflows, as they can use the same JTAG controller to access embedded instruments. By extending the capabilities of the traditional JTAG standard, IJTAG allows designers to manage embedded instruments more efficiently, without having to develop custom access methods for each new design.
IJTAG provides a standardized framework for accessing and controlling a variety of embedded instruments. This simplifies the design and integration of instruments such as BIST engines, sensors, and monitors into an SoC, enabling better test coverage and easier diagnostics. With the use of ICL and PDL, IJTAG allows for the automatic generation of test procedures and access methods for embedded instruments. This increases the level of automation in the test development process, reducing the time and effort required for test generation. The ability to dynamically reconfigure the instrument network means that IJTAG can optimize test access paths, reducing the time taken to reach and test specific instruments. This leads to faster test execution, especially in large, complex SoCs. IJTAG enables precise control and configuration of instruments that monitor the operational health of a chip. This allows for better diagnostics, helping designers pinpoint specific areas of failure or performance issues during production testing or field operation. As SoC designs scale up in size and complexity, IJTAG provides a scalable and flexible architecture to manage increasing numbers of embedded instruments, without significant increases in test overhead. By integrating with the existing JTAG infrastructure, IJTAG does not

require additional interface pins or specialized hardware, making it a cost-effective solution for accessing and managing embedded instruments. [8]

1.1.3. Tessent

Tessent is a comprehensive suite of semiconductor test and yield analysis solutions provided by Mentor, a Siemens business. One of its most powerful capabilities lies in Memory Built-In Self- Test (MBIST), which is used to automate the testing and diagnosis of embedded memories within integrated circuits (ICs). Tessent MBIST tools enable designers to implement, verify, and analyze built-in self-test and repair features for embedded memories like SRAM, DRAM, ROM, and flash. These tools are critical in ensuring high reliability and fault coverage while minimizing test costs and efforts during production.

Tessent's MBIST solution is designed to ensure that, High fault coverage is achieved to identify various types of memory faults such as stuck-at faults, transition faults, and coupling faults. In- field testing and repair of memory faults are facilitated. Testing is optimized for both speed and efficiency, particularly in modern, highly complex SoC designs. Key Features of Tessent MBIST are as listed:

Tessent provides tools to automatically generate MBIST logic and test patterns for a wide range of memory types. The generated MBIST ensures thorough testing by applying various memory fault models and patterns, such as March algorithms, to detect defects effectively. The tool optimizes the generated test patterns to minimize testing time and silicon overhead, thus ensuring that the embedded MBIST does not significantly impact the overall performance of the chip.

One of the standout features of Tessent MBIST is its ability to integrate seamlessly into existing semiconductor designs. It generates the necessary MBIST circuitry with minimal impact on the device's area, power consumption, and performance. Tessent MBIST logic can be added to the design without requiring major modifications, preserving the original functionality of the chip while adding comprehensive test capabilities [2].

Tessent provides advanced fault simulation capabilities, allowing designers to evaluate the effectiveness of MBIST patterns. Fault simulation helps in assessing the test coverage of MBIST for various fault models, enabling designers to make sure

that all potential failure modes of the memory are detected. The simulation environment allows verification engineers to identify gaps in fault coverage and to fine-tune test patterns, ensuring that memory defects will not escape during production testing.

In addition to detecting faults, Tessent MBIST includes robust memory repair mechanisms. Tessent's Built-In Repair Analysis (BIRA) or Built-In Redundancy Analysis (BIRA) tools allow for automatic identification of faulty memory locations and their replacement with redundant memory rows or columns [3]. This repair feature helps improve manufacturing yield by allowing chips with minor defects to be repaired and used rather than discarded, significantly reducing production waste and cost. Tessent supports both soft (in-field programmable) and hard (fused) repairs, giving designers flexibility depending on the application's requirements.

Tessent MBIST incorporates power-aware test capabilities. In modern semiconductor designs, power management is critical, and testing must account for different power modes (e.g., sleep mode, power-down mode). Tessent allows memory to be tested under various power conditions, ensuring that low-power features do not introduce new defects. Power-aware testing reduces power consumption during testing and ensures that memory elements in power-gated or clock- gated domains are properly tested and verified.

Tessent MBIST provides comprehensive diagnostic tools to pinpoint the exact location and type of memory faults. Diagnostic capabilities allow designers to identify the root cause of memory failures, leading to faster debug and more targeted yield improvement efforts during production. This feature is particularly useful for design teams trying to improve yield and address specific failure modes in a memory-intensive SoC.

Tessent allows users to implement custom MBIST algorithms tailored to specific memory architectures or performance needs. While standard algorithms like March C and March X are widely supported, users can also define custom test strategies to handle unique memory configurations and fault models. Customizable algorithms enhance flexibility and enable designers to focus on specific faults or performance targets.

Tessent MBIST supports various types of memories, including: SRAM, DRAM, ROM, Flash and embedded memories. [3]

1.1.4. VCS

VCS is a high-performance simulation and verification tool commonly used in digital design and verification processes. It is developed by Synopsys, a leading provider of electronic design
automation (EDA) software and services. VCS is widely utilized for simulating and verifying digital designs described in hardware description languages (HDLs) like Verilog, System Verilog, and VHDL.

VCS is known for its high simulation speed and efficiency, making it suitable for complex and large-scale designs, it uses advanced algorithms to optimize simulation performance, allowing faster execution of testbenches and verification tasks. This is crucial in modern IC design, where verifying huge systems-on-chip (SoCs) can take a significant amount of time. VCS supports multiple hardware description languages, including Verilog, System Verilog, and VHDL, making it versatile for use in various design environments. This allows engineers to verify designs using their preferred or required HDL.

It provides powerful debugging capabilities, it integrates with Synopsys' waveform viewer, like DVE (Design Visualization Environment), to provide a detailed view of the simulation results. Engineers can visualize signal transitions and relationships, helping them pinpoint issues in the design. VCS allows for interactive debugging during simulation, where engineers can step through the code, monitor signal values, and control the simulation process. This real-time interaction helps in isolating and fixing issues more efficiently. The tool provides a way to trace the behavior of specific signals through different parts of the design, making it easier to track down the root causes of functional or timing issues.[7]

VCS supports System Verilog Assertions (SVA), a key feature in assertion-based verification. Engineers can embed assertions directly into their code to monitor for design violations during simulation. This helps in catching and isolating bugs early in the verification process. VCS provides support for functional coverage, enabling comprehensive analysis of the verification process. Engineers can define functional coverage points to ensure that the design has been thoroughly tested for all possible scenarios and corner cases. This ensures design correctness and completeness of the verification process. VCS fully supports UVM (Universal Verification

Methodology), which is a standardized framework for creating reusable and scalable verification environments. UVM allows engineers to build modular testbenches that can be reused across different projects, improving efficiency and reducing verification time. VCS is highly optimized for System Verilog, which is commonly used in advanced verification methodologies. It supports features such as constrained random stimulus generation, coverage-driven verification, and object-oriented testbench creation. For legacy projects, VCS also offers strong support for VHDL simulation.

VCS has the capability to verify mixed-signal designs, integrating analog and digital simulation. This feature is essential for verifying complex systems that combine digital logic with analog components, such as ADCs, DACs, or mixed-signal SoCs. VCS includes features for power- aware simulation to verify the functionality of low-power designs, ensuring that power management strategies like clock gating, power gating, and voltage scaling are implemented correctly and do not introduce functional errors. VCS allows running large-scale regression tests efficiently. It supports parallel simulation, leveraging multi-core processors and distributed computing environments to run multiple simulations concurrently. This improves throughput and reduces the time to complete large verification tasks. [7]

1.1.5. Verdi

Verdi is a sophisticated debugging and visualization tool used in the field of digital design and verification. It is developed by Synopsys, a leading provider of electronic design automation (EDA) tools. Verdi is particularly well-suited for debugging complex digital designs described in hardware description languages (HDLs) such as Verilog, System Verilog, and VHDL. The tool is designed to provide engineers with advanced debugging capabilities and enhanced visualization of simulation results.

Verdi offers a user-friendly graphical environment that allows engineers to debug and analyze their digital designs efficiently. It supports transaction-based debugging, enabling engineers to trace and analyze transactions in their designs at a higher level of abstraction. Verdi integrates waveform and source code views, providing a comprehensive understanding of the design's behavior. Engineers can visualize the hierarchy of their designs graphically, making it easier to navigate and understand

complex structures. Verdi supports the analysis of functional coverage data, allowing engineers to assess the completeness of their verification efforts. It provides insights into code coverage, helping identify untested portions of the design. [4]

1.2 Motivation

Since deep sub-micron devices have multiple memories that require a small footprint and quick access times, an automated test method is necessary to minimize ATE time and expense. The need to test memory defects and its capacity for self-repair is not entirely covered by conventional DFT approaches. MBIST, which incorporates test and repair circuitry into the memory itself and offers a respectable yield, offers a viable answer to this problem. The need to test memory defects and its capacity for self-repair is not entirely covered by conventional DFT/DFM techniques. MBIST, which incorporates test and repair circuitry into the memory itself and offers a respectable yield, offers a viable answer to this problem. The purpose of this paper is to enlighten readers about the MBIST architecture, several memory fault models, algorithmic testing of such models, and memory self-repair mechanisms.

1.3 Problem Definition

To understand the role MBIST, it's validation and memory repair in VLSI Flow and perform the process on the given IP partition. The challenge is to develop a comprehensive and efficient memory testing and repair framework that can identify and rectify memory faults in a variety of scenarios. This includes addressing manufacturing defects, wear-out effects, and intermittent faults to enhance overall memory reliability, to develop methods for detecting manufacturing defects, such as stuck-at faults, bridging faults, and pattern-sensitive faults, to explore repair mechanisms for correcting identified faults, including redundant element activation, error correction codes (ECC), and spare cell utilization, to optimize testing and repair algorithms to minimize the time required for memory diagnostics and correction, ensuring minimal impact on overall system performance.

1.4 Objectives

- Understanding memory testing and memory repair importance in VLSI design flow.
- Inserting and validating MBIST logic on the given IP partition.
- Generating MBIST TAPs and validating them.
- Generating different kind of patterns for SOC level validation

.

CHAPTER-2

LITERTAURE REVIEW

Design for Testability (DFT) refers to the incorporation of specific design techniques that improve the ability to test and diagnose integrated circuits (ICs) after they are manufactured. DFT is critical in modern chip design due to the complexity and miniaturization of electronic systems, which makes it challenging to ensure fault-free functionality in devices. DFT is primarily focused on identifying potential faults during the testing process. Common fault models include:

- Stuck-at Faults: A line in a circuit is permanently stuck at a logic high ('1') or logic low ('0').
- Transition Faults: These faults model failures in signal transition from '0' to '1' or vice versa.
- Bridging Faults: Occur when two signal lines in a circuit are mistakenly connected, leading to a short-circuit.
- Delay Faults: Faults that cause signals to arrive late, resulting in timing failures in high-speed circuits.

Test patterns must be generated to detect these faults effectively. These patterns are designed to expose faults by driving inputs and observing outputs. BIST is a DFT technique where testing capabilities are embedded within the chip itself, allowing for automated testing without external hardware. BIST is often used for memory, logic, and complex systems. A common DFT technique that involves connecting all flip-flops in a circuit into one or more long shift registers (scan chains). This allows for easy shifting in of test vectors and reading out test results. Defined by the IEEE 1149.1 standard, boundary scan allows for testing of interconnections between chips in a circuit board without using physical test probes. It's particularly useful for highly dense boards with fine-pitch components. DFT designs aim to maximize the number

of faults that can be detected and diagnosed. High fault coverage is essential to ensure that defects are detected before products ship to customers. [1]

Even though DFT has many advantages in chip testing and manufacturing it also has many challenges such as stated. As chip designs become more complex, the number of possible fault scenarios grows exponentially. Ensuring high fault coverage requires sophisticated test pattern generation and may increase test time and cost. Adding DFT structures (e.g., scan chains, BIST) can increase the chip area and complexity, potentially impacting performance and power consumption. A more comprehensive test procedure can significantly increase the time required to test each chip, which can lead to higher costs, especially in large-scale production. Timing faults and small delay defects, particularly in high-performance circuits, are challenging to detect and often require additional testing techniques like at-speed testing. [2] The share of embedded memories in systems-on-a-chip (SOCs) is growing. Memory is dispersed across the device as opposed to being concentrated in one area. Memories come in a variety of sizes and forms. It is challenging to test access to these memories via a small number of chip input/output pins. Memories can occupy a significant amount of space in a lot of designs nowadays. Because embedded memory yield determines chip yield, checking embedded memories using a pass/fail system is insufficient. A Design for Testability (DFT) technique called Memory Built-In Self-Test (MBIST) uses a portion of a chip, board, or system's circuit to test the circuit itself. The test vector for memory testing is provided by the FSM (Finite-State Machine) in the BIST controller itself. Thus, there is there is no need for external test vector for testing. [3]

Memory Built-In Self-Test (MBIST) has become a critical component of Design-for-Testability (DFT) in modern SoCs due to the rapid growth of embedded SRAM density and the increasing complexity of deep-submicron manufacturing processes. As SoCs integrate hundreds of distributed SRAM blocks operating at high frequencies, external testing becomes impractical, making MBIST insertion an essential step in ensuring yield, reliability, and cost-efficient test execution.

A fundamental motivation for MBIST insertion is that SRAM arrays exhibit high sensitivity to manufacturing defects due to dense layout structures, complex bit-cell

interactions, and high operational speeds. Zhang *et al.* [9] highlight that SRAM often occupies over 70–90% of SoC area, making memory testing the dominant factor in overall chip test quality. Their implementation reused the SoC's JTAG interface and on-chip PLL to perform at-speed MBIST on 677 SRAM blocks grouped into 36 controllers. They show that low-speed MBIST fails to detect transition faults, whereas at-speed MBIST can identify memory failures that manifest only at functional frequency. This establishes the necessity of incorporating at-speed capabilities during MBIST insertion.

In large SoC designs with thousands of SRAM instances, a key design challenge in MBIST insertion is how memories are grouped under MBIST controllers. Inefficient grouping increases area overhead, routing complexity, and test time. Yang *et al.* [10] present an efficient grouping strategy that divides the MBIST insertion flow into hard-constraint grouping (clock domain, interference restrictions) followed by soft-constraint optimization (power limits, distance limits). Their hybrid Greedy + Simulated Annealing method significantly reduces the number of controllers and offers better power and routing optimization compared to traditional ILP or heuristic-only approaches. This demonstrates that MBIST insertion today is not only about algorithm choice but also about scalable controller planning.

MBIST insertion must also integrate smoothly with the broader SoC DFT infrastructure. Jois and Ravish [11] emphasize the need for IJTAG-compliant registers, Segment Insertion Bits (SIBs), scan chains, EDT decompressors, and on-chip clock controllers for proper test access and observability. Their work further stresses the requirement for secure access to test networks, introducing an authentication-based protection mechanism before enabling MBIST or other DFT components. Their findings underline that MBIST insertion is deeply tied to JTAG/IJTAG planning, with SIB networks determining which memory blocks are activated for test, and with secure configurations preventing misuse of test paths in delivered silicon.

CHAPTER-3

METHODOLOGY

3.1 MBIST Architecture

Tessent MBIST provides a complete solution for the at-speed testing, diagnosis, repair, debugging, and characterization of embedded memories. The MBIST Architecture as shown in Figure. 3.1 consists of memories, where each memory is connected to its memory interface. These memory interface may have BIRA logic or may not depending on the memory. Repairable memories implement spare redundant rows or columns which enables the memory repair by directing the faulty rows or columns to the spare ones. If the memory is repairable, the interface consists of BIRA logic which can determine whether the memory is repairable, and its repair information based on the specified redundancy scheme.

The repair solution is stored in BISR registers. All the BISR registers on the chip are connected to form BISR Chain. The BISR chain is connected to chip level BISR controller depending on the location of the Fuse box the interface signals between the Fuse box and the BISR controller is determined. The BISR controller compresses the content that is scanned out of BISR chain and writes the compressed data into the Fuse box.

Figure 3.1 MBIST Architecture [1]

The BISR controller can also decompress the repair data in the Fuse box and scan in the BISR chain. At a higher level there is a central TAP which is connected to the BIST engines which are scattered across the chip, The TAP is connect to BIST engines through BAP. TAP is also connected the BISR controller to access the repair information. [8]

3.2 MBIST Insertion

MBISTArchitect can be invoked in two phases: BIST insertion phase and BIST generation phase.

- BIST Insertion Phase

In the BIST Insertion phase, you can create BIST logic and insert that logic into your design, or you can insert already generated BIST into your design.

- BIST Generation Phase

In the BIST Generation phase, you only generate BIST logic. You would use the BIST insertion phase to insert the logic you generated in this phase.

The BIST insertion phase has three modes: SETUP, BIST, and INT.

SETUP Mode: In the Setup mode the screen prompt is SETUP> and all SETUP legal mode commands (commands like Load Library, and Load Design Object) are available, and can be used.

BIST Mode: In the BIST mode the screen prompt is BIST> and all BIST legal mode commands (commands like Add New Controller, and Add Existing Controller) are available, and can be used.

INT Mode: In the Integration mode the screen prompt is INT> and all INT legal mode commands (commands like Add Pattern Translation and Delete Patterns) are available.

Switching between the above-mentioned modes is allowed using the set system mode command. The tool has three flows related to the insertion phase: top-down, bottom-up, and block based.

Top-Down Insertion Flow

We can complete the BIST generation and insertion tasks in a single tool invocation when using the top-down insertion approach. BIST controllers are generated first, then they are inserted into the netlist and connected to the top-level design. The BIST circuitry is saved to intermediate files as a result of the generation activity, along with a controller test bench that only drives those files. The top-down insertion flow chart is displayed in Figure 3.2.

A corrected version of the input chip or top HDL with the BIST circuitry inserted and connected is saved as a result of the insertion activity. Nothing about the original design or signal names is altered in the BIST-inserted output files the file structure of the input chip or top HDL is preserved. Lastly, you can complete the integration activity in this flow, which yields a top-level test bench that uses the integration patterns to drive the output netlist that has been inserted into the BIST. Afterwards you can verify the generated/inserted outputs by simulating the test benches.

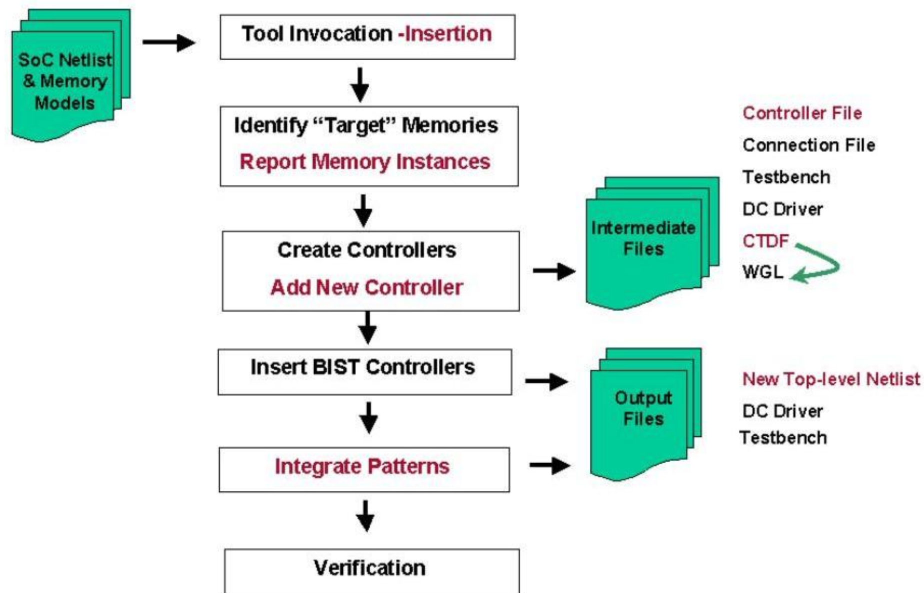

Figure. 3.2 Flow chart of Top-Down insertion flow [5]

Figure. 3.3 shows the circuit before BIST insertion using the top-down flow, where we can see there are no controllers that has been added whereas Figure. 3.4 shows the same circuit after MBIST insertion using the top-down flow, in which we can see the BIST controller which is referred to as Cont in the Figure. 3.4 has been added. Also, BIST inserted memory collars that surround the memory after the insertion which is not shown in the Figure. 3.4.

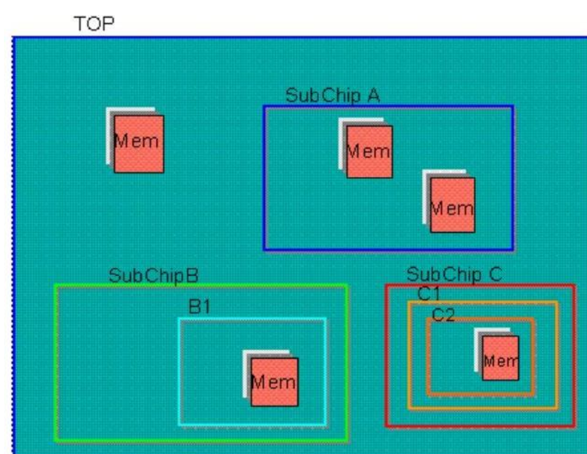

Figure. 3.3 Circuit before BIST insertion [5]

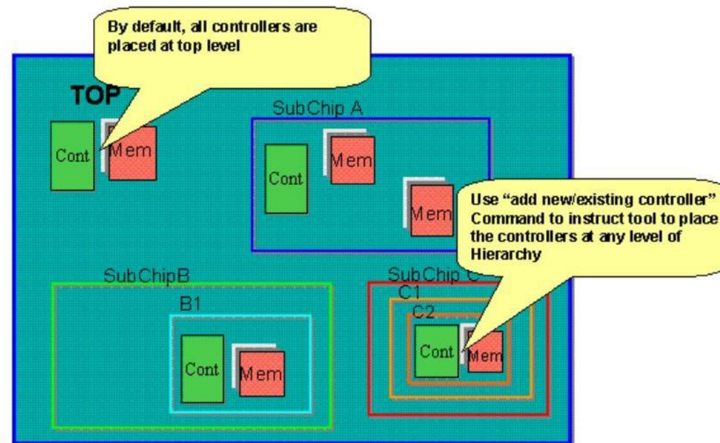

Figure. 3.4 Circuit after BIST insertion [5]

In the bottom-up insertion flow, the activities of BIST generation and BIST insertion are performed in two separate invocations of the tool.

Block-based insertion flow is used where two or more teams are designing the chip in blocks. Using a block-based flow, each team can perform their BIST activities separately. Also, when performing the insertion activity on extremely large chip designs will need a powerful workstation to run the tool. If the CPU and workstation memory resources are limited, we can simplify the insertion activity by using the Block-Based Insertion Flow, which involves multiple invocations of the tool and breaks the large task of insertion into several smaller subtasks. The steps are as follows,

BIST generation and insertion activity on a single design block. Repeat this step for all major memory related blocks in the design.

BIST insertion activity on the top-level design, unifying the various design blocks of the previous step.

MBIST Architect works from a library model, as well as a design netlist to do the insertion. The library model, along with a small set of application commands, is all the tool requires to generate the appropriate BIST circuitry. Below lists the set of files the tool requires:

Chip design files: These are the RTL design files for the chip or top level or the block level design files which depends on the insertion that is being used.

Library files: These files describe the memory model for the BIST generation activity; the memory model provides all the information which are required by the tool to create BIST controller.

Do file: This file includes list of valid MBIST architect commands.

Config file: this includes all the collaterals which is specific to product. It contains clock information, different switch options that are specific to partition, type of memory and controller for memory instances.

Following MBIST insertion, a series of algorithms is used to validate the partitions. The majority of the occurring memory errors have been shown to be detected by these memory test algorithms. Because the hardware required to generate the patterns is tiny and able to service numerous on- chip memories, several of these algorithms are ideally suited for BIST.

The March tests are the most widely used algorithms. When values are written to and read from known places in memory, March tests produce patterns that "march" up and down the memory addresses. The memory size and word length can be automatically calculated by these algorithms by retrieving the appropriate values from the memory model.

3.3 IEEE 1687 IJTAG Architecture

The purpose of IEEE 1687 Internal JTAG (IJTAG) is to simplify the use of embedded instruments. The goal is to make it easier to implement these embedded instruments in a broader range of validation, testing, and debugging applications at the chip, board, and system levels. Chip and board levels will use the IEEE 1687 IJTAG standard. When

IJTAG is used in conjunction with a simulator or emulator for design verification, it can be particularly helpful for chip designers. Additionally, IJTAG will be implemented in system test environments and ATE test environments, where it will be used in yield analysis, chip characterization, chip debug/diagnostics, and chip testing. [8] The three main functional domains of this IJTAG ecosystem are Creation, Integration, and Use.

The ’creation’ phase generally involves IP providers and IC logic designers who will design or use EDA tools, such as Verilog generators. ‘Integration’ will encompass collecting the IJTAG Register-Transfer Level (RTL) data and intellectual property (IP) from IP creators and placing this IP into chips. IC integrators will then perform verification, power analysis, timing closure, layout optimization, and other tasks. Integration will also include bringing together all the IJTAG

Instrument Connectivity Language (ICL) and Procedure Description Language (PDL) files that will complete the bundle of embedded instrumentation IP and ensure its portability into multiple chip designs. Once a chip including IJTAG IP is fabricated, IC test, debug and yield analysis processes will begin the ‘use’ phase at the chip level. [11]

Figure. 3.5. illustrates an IEEE 1687 IJTAG architecture at the chip level. The right side of the drawing shows the IJTAG network interfacing to IJTAG-compliant embedded instruments. The IEEE 1149.1 boundary-scan (JTAG) standard’s Test Access Port (TAP) is on the left. The TAP functions as the interface for the embedded 1687 IJTAG architecture to the world outside of the chip. Essentially, the boundary-scan TAP and its TAP Controller can access the embedded IJTAG instruments by accessing and operating the IJTAG network that connects the controller to the instruments. Several other IJTAG concepts are shown in this illustration, including the Segment Insertion Bit (SIB), ICL and PDL.

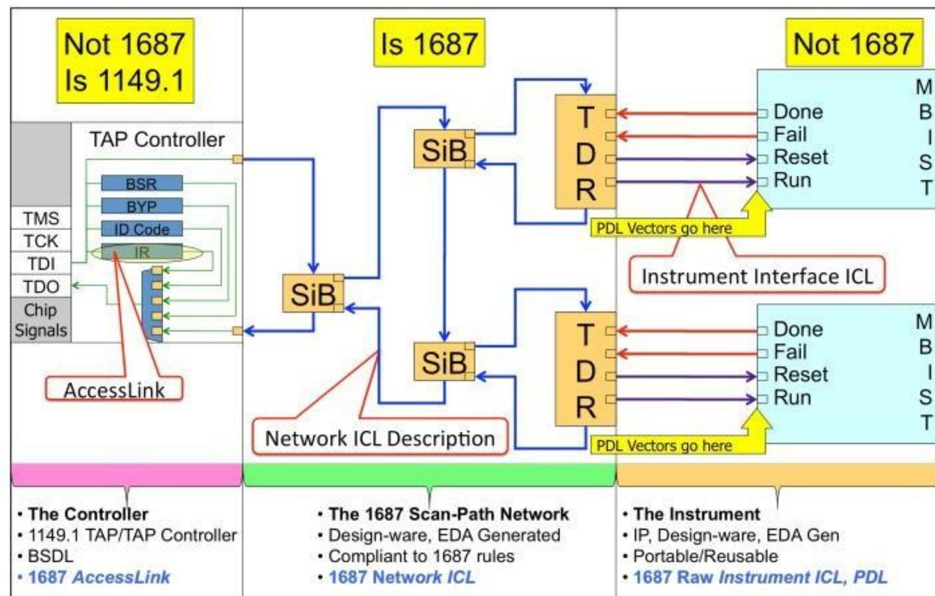

Figure. 3.5 IEEE 1687 IJTAG architecture [8]

The left side of Figure. 3.5. shows a JTAG controller that interfaces the IJTAG network with embedded instruments to the chip's pins. This controller generates the operating protocol for the embedded instrument access network. The IEEE 1687 IJTAG standard does not define this controller, but it does describe how the IJTAG network is connected to any controller through a mechanism known as an Access Link, which is defined in the standard. Currently, the only controller referenced in the 1687 IJTAG standard is the IEEE 1149.1 boundary-scan (JTAG) standard's Test Access Port (TAP) and TAP Controller. However, the IEEE 1687 IJTAG standard's Access Link allows for other controllers, such as a direct pin interface or other controllers besides the JTAG. Other controllers in addition to the JTAG controller could be defined in future extensions to the IEEE 1687 IJTAG standard.

Regardless of controller type, such as JTAG, direct chip pins, SPI, I2C or others, the controller must provide the driving control and data interface signals to enable the IEEE 1149.1 JTAG operations of Shift, Capture, Update, and Reset for the JTAG-like Test Data Registers (TDR) which comprise the IJTAG on-chip network.

On the right side of Figure. 3.5. is an embedded instrument. The composition of these embedded instruments is not defined in the IEEE 1687 IJTAG standard because the framers of the standard did not want to limit how instruments should be made by incorporating instrument restrictions into the standard itself. The only portion of an instrument that is defined in the IEEE 1687 IJTAG standard is the description of the signal interface connected to the instrument's TDR on the IJTAG instrument access network.

IJTAG embedded instruments are self-contained blocks of functionality. In addition to the interface to the IJTAG network, some instruments may also include the targets of their functionality. For example, a temperature monitoring instrument may also include a temperature sensor. The instrument controls its target, which in this case is the temperature sensor. The other alternative is for an IJTAG instrument to be composed of its interface to the instrument network (TDR) and the operational protocol for the target, but the target remains a separate standalone entity. An example of such an instrument might be a memory test instrument (memory BIST in Figure. 3.6) that is separate and independent from the memory itself. In addition to the memory test instrument, other instruments or other functionalities within the system could perform operations on the same memory. So, the IJTAG embedded instrument might be one of many units that would access and operate on the target.

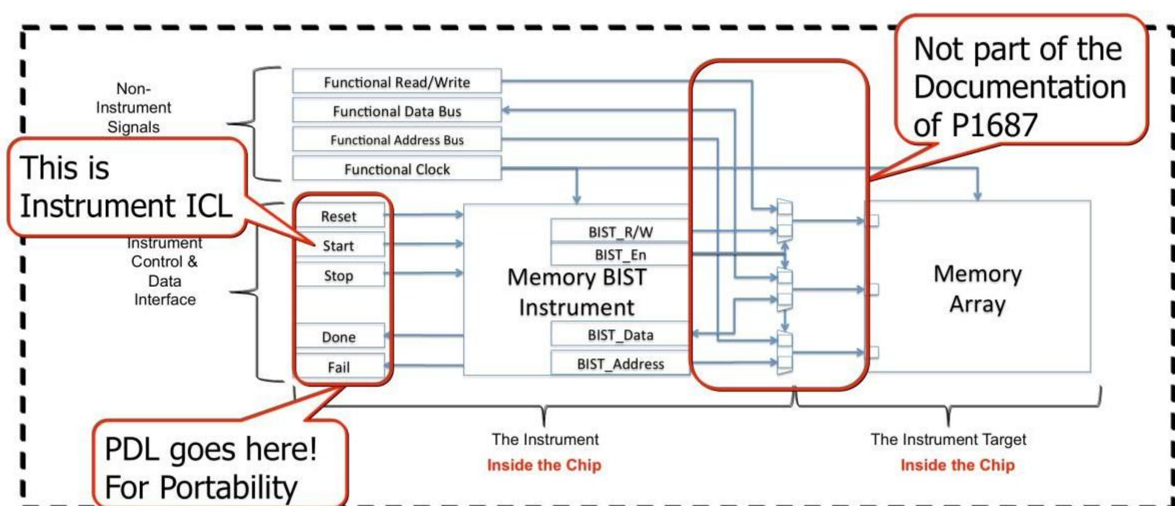

Figure. 3.6 An example embedded instrument showing location of PDL which enables IP portability [8]

An IJTAG network connects embedded instruments to the network controller. The goal of the architecture description in the IJTAG standard is to allow options, tradeoffs, and optimizations to be applied to the IJTAG network so the network may support operation and engineering tradeoffs, and so that network segments may have a measure of plug-and-play portability. The IJTAG network is made of serial scan path bits that can be organized as either of two different types of objects: 1) Test Data Registers (TDR); or 2) Segment Insertion Bit (SIB). Although the IEEE 1149.1 boundary-scan JTAG standard refers to an entire scan path within a chip as one TDR, within the context of the IEEE 1687 IJTAG standard the scan path connecting embedded instruments is more accurately described as a segment of the chip's overall JTAG scan path which is made up of the individual TDRs that interface to each embedded instrument on the scan path. As a result, this IEEE 1687 on-chip IJTAG network is usually described as being comprised of tens or even hundreds of TDRs. TDRs are viewed as data bits associated with embedded instruments.

An entire network or subsections of an IJTAG network of embedded instruments should always maintain a compliant separable signal interface. If a network is divided for some reason, the same separable interface should provide access to both subdivided segments of the original network. This separable 1687 IJTAG interface is what makes an IJTAG network portable. For example, an embeddable IP core might contain a whole and complete IEEE 1687 IJTAG

embedded instrument access network with multiple embedded instruments. When this core is integrated into a chip, the core’s IJTAG network and connected instruments can be easily integrated within a larger IJTAG access network that already exists on the chip by connecting the core’s IJTAG separable network interface to the control signals provided by the chip’s TAP that operates the IJTAG network on the chip.

One of the main purposes for IEEE 1687 IJTAG is to provide effective management processes and procedures for the wealth of embedded instruments that are being integrated into today’s semiconductor devices. This can mean enabling a variety of capabilities, including the following:

- Modifying the length of the scan path
- Flexibly scheduling instrument operations
- Operating multiple instruments concurrently, reducing test execution times or instrument operation times for multiple instruments
- Coordinating instrument operations
- Other types of instrument activities yet to be defined

The SIB (Segment Insertion Bit) is one of the fundamental components specified in the IEEE 1687 IJTAG standard. Comparing a SIB to an IEEE 1149.1 boundary-scan shift/update cell reveals that the former dynamically configures an on-chip 1687 IJTAG scan path to satisfy a specific set of test vector requirements. A section of the chip's IJTAG scan path can be activated by "selecting" a particular SIB, which will then activate the instrument(s) on that section of the scan path. In contrast, "de-selecting" a SIB will make the instruments on that segment inaccessible and disable a section of the chip's overall scan route. Instruments on a deactivated segment of the scan path cannot be accessed as long as the scan path segment is deactivated, but they can still hold a state that operates an embedded instrument while their network segment is deselected (the rule is that a deselected scan segment must hold state for all operations except reset). This feature allows an instrument to be started, then sequestered away from other instruments while other operations are being conducted on the active scan path (as opposed to parking in the Run-

Test-Idle state until the instrument has completed its function). The overall effect is that the length and composition of the scan path is dynamic.

3.4 TAP Creation

In a SoC there can be many such MBIST inserted IPs which will need to be connected to the Test Access Port (TAP) network for the SoC. The integration flow allows a user to specify one or more MBIST inserted IPs which they would like controlled by a single TAP and performs the following steps to integrate the IPs together with a TAP controller.

1. Integration flow creates a pseudo-SoC Verilog wrapper in which all logic to be connected together will be placed. This wrapper is only present to provide a structure for the tool to operate on and will not be used by the SoC.
2. Instantiates the MBIST Inserted IPs to be connected to the TAP within the pseudo-SoC Verilog wrapper.
3. Creates a Verilog wrapper used to encapsulate any TAP related logic into a single module. This wrapper is referred to as the "shell".
4. Instantiates TAP logic which will be used to control the MBIST inserted IPs within the pseudo-SoC Verilog wrapper. Alternately, the TAP logic can be generated by the tools during the integration step itself.
5. Queries the TSDB for each MBIST inserted IP to determine what MBIST capabilities exist and what signals will need to be connected to the TAP controller.
6. Builds a blueprint for the TAP/MBIST network it will create (called a DFT Specification).
7. Creates connections between TAP controller and the BAP of the MBIST inserted IPs per DFT specification. If a TAP logic needs to be generated it is done during this step.
8. Validates the TAP/MBIST network by ensuring it meets all MBIST requirements for each IP outlined in the TSDB files for each IP, creates a new TSDB for the complete TAP/MBIST network, generates an IEEE 1687 Instrument Connectivity Language (ICL) file for the TAP/MBIST network, and creates Intel specific information files which will be used by downstream integration tools to instantiate the TAP/MBIST network into the larger SoC TAP network.

3.5 Pattern Generation and Verification

Tessent MemoryBIST uses a similar flow for generating patterns at the block, core, or top level. You typically generate only Verilog testbenches and not manufacturing test patterns at the block

or core level, because that portion of the circuitry normally gets instantiated in a larger top-level design.

At the top level, you generate Verilog testbenches before signing-off the design. You then create Automated Test Equipment (ATE) test patterns, to apply as manufacturing tests on silicon devices.

In bottom-up flows, MBIST-inserted lower-level blocks or cores are fully verified in a standalone fashion before being integrated into a larger design. This hierarchical DFT method enables completing MBIST insertion in smaller design portions, even though the rest of the design may not be ready yet. Because MBIST only requires valid clocks and a low-speed serial test access to run, verifying a block or core after it has been integrated into the full (top) design is greatly simplified.

In top-down flows, MBIST insertion is done across the entire design at once. This methodology requires fewer steps overall than a hierarchical DFT insertion, however the MBIST insertion has to be verified across the entire design before it can be signed-off. This process consequently takes longer to perform and requires more computing resources than a hierarchical approach. This full-chip verification task typically happens at a very critical time (for example, when the chip is essentially completed and just about to tape out), so unexpected delays may impact the design schedule.

The figure. 3.7 illustrates the pattern generation for MemoryBIST controllers that are inserted in a core or top design. We generate the following:

- For the bottom-up flow, core-level Verilog testbenches
- For the top-down flow, top-level manufacturing test patterns and Verilog testbenches

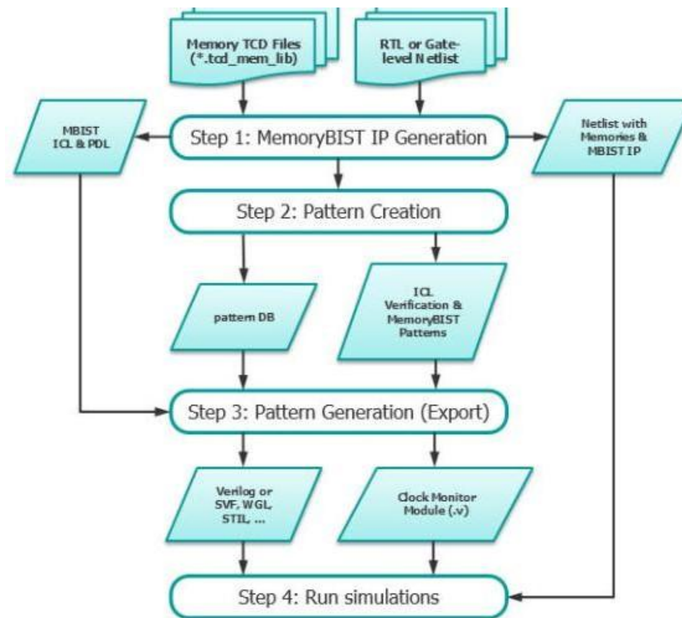

Fig 3.7 Pattern Generation Flow [5]

3.5.1 Creating simulation testbench patterns

Use this procedure to generate Verilog simulation testbench patterns that can be applied at the block, core, or top level.

Signoff, or simulation testbench patterns, are generated by default. The test patterns only target controllers inserted in the current design. Controllers in lower level physical blocks can be included when the property simulate_instruments_in_lower_physical_instances is set to on in the DefaultsSpecification. All controllers are run in parallel. Additionally, for controllers with RAMs you can limit testing to address corners by setting the patterns specification property reduced_address_count to on.

The required inputs for this step of the flow that you specify from the tool prompt or within the pattern generating dofile are as follows:

- The TSDB (the tsdb_outdir) of a completed block, core or top-level design.
- Extracted ICL for the current design portion (normally found in the TSDB).
- Exported PDL with procedures for the current design (also found in the TSDB).
- An RTL or gate-level netlist of the current design for sign-off simulations.

The following procedure assumes the current design already went through MBIST insertion. Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

% tessent -shell

2. Set the tool context to IJTAG patterns mode using the set_context command as follows: SETUP> set_context patterns -ijtag

3. Open the TSDB with the open_tsdb command, if it is not already open (which would be the case if you just completed MBIST insertion within the very same Tessent Shell session). For example:

SETUP> open_tsdb tsdb_outdir

4. Unless it is already in memory, read the current design's extracted ICL and TCD with the read_design command. For example:

SETUP> read_design blockA -design_identifier rtl -no_hdl

5. Set the current design with the set_current_design command. For example: SETUP> set_current_design blockA

The TSDB is analyzed and all inserted DFT logic (including MBIST controllers) are identified.

6. Create a pattern specification using the create_patterns_specification command. The example below sets a variable with the identification for the created specification, which can be used in modifying the specification if needed.

SETUP> set pat_spec [create_patterns_specification] Memory BIST patterns are generated at this point.

7. If wanted, follow this step to set reduced_address_count to off for testing the full address range of memories. Otherwise, continue to the next step. The process outlined can be modified to configure other properties within the pattern specification as needed.

a. Specify the path to the wrapper containing the property. For example:

SETUP>set

    wrap

    [get_config_elements

    -in_wrapper

    ${pat_spec}

    \ Patterns(MemoryBist_P1)/TestStep/MemoryBist]

The variable $wrap now contains the full path.

b. Set the reduced_address_count property to the off setting. For example: SETUP>set_config_value reduced_address_count \ -in_wrapper ${wrap} off The patterns specification is now updated with the wanted property setting.

c. Optionally, confirm the setting by inspecting the patterns specification: SETUP>report_config_data $pat_spec

The patterns specification is displayed for examination.

8. Process the pattern specification:

SETUP> process_patterns_specification

9. Point to the simulation library sources so all design files can be found. For example: SETUP> set_simulation_library_sources -y ./memlibs -extensions { v }

10. Simulate the memory BIST testbenches with the following command: SETUP> run_testbench_simulations

11. The above simulation can be monitored or checked with the following command: SETUP> check_testbench_simulations

3.5.2 Creating manufacturing test patterns

Use this procedure to generate manufacturing test patterns that can be applied at the top level. Generated patterns are grouped based on the number of asynchronous ATE clocks that can be active within one pattern. The number is based on the DefaultsSpecification max_async_clock_sources property that defaults to unlimited, placing all memory BIST controllers inside of a single pattern, operating in parallel and potentially belonging to different asynchronous frequency groups.

Generated patterns can be further split into multiple test step or procedure step pattern files, to enable test program events such as VDD bumping or PMU measurements being inserted in the middle of the patterns. Refer to the AdvancedOptions/split_patterns_file property descriptions in the TestStep and ProcedureStep wrappers for more information about patterns file splitting.

Manufacturing test patterns always perform tests across the full memory address space. The patterns specification property reduced_address_count defaults to off, and unlike signoff patterns, it is not set to on when a controller is testing RAMs only.

Manufacturing pattern files can be generated in WGL, STIL, STIL2005, TITD, FJTDL, MITDL, TSTL2 and SVF formats. The format you want can be specified with the PatternsSpecification manufacturing_patterns_formats property. When unspecified, the STIL format is generated. For a description of these formats, see the description of the format_switch argument of the write_patterns command. The required inputs for this step of the flow that you specify from the tool prompt or within the pattern generating dofile are as follows:

• The TSDB (the tsdb_outdir) of a completed top-level design.

• Extracted ICL for the current design (normally found in the TSDB).

• Exported PDL with procedures for the current design (also found in the TSDB). Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

% tessent -shell

2.Set the tool context to IJTAG patterns mode using the set_context command as follows: SETUP> set_context patterns -ijtag

3.Open the TSDB with the open_tsdb command, if it is not already open (which would be the case if you just completed MBIST insertion within the very same Tessent Shell session). For example:

SETUP> open_tsdb tsdb_outdir

4.Unless it is already in memory, read the current design's extracted ICL and TCD with the read_design command. For example:

SETUP> read_design blockA -design_identifier rtl -no_hdl

5.Set the current design with the set_current_design command. For example: SETUP> set_current_design top

The TSDB is analyzed and all inserted DFT logic (including MBIST controllers) are identified.

6.Create a pattern specification using the create_patterns_specification command. The example below sets a variable with the identification for the created specification, which can be used in modifying the specification if needed.

SETUP> set pat_spec [create_patterns_specification manufacturing]

Specifying the manufacturing usage option automatically adds the usage : manufacturing_test property to the PatternsSpecification. Memory BIST patterns are generated at this point.

7.If you want, follow this step to specify the wanted pattern file format with the manufacturing_patterns_formats property in the PatternsSpecification. If the default STIL format is acceptable, continue to the next step. The process outlined can be modified to configure other properties within the pattern specification as needed.

a.Using the variable set in Step 6, set the manufacturing_patterns_formats property to the wanted setting. For example, for WGL format:

SETUP>set_config_value manufacturing_patterns_formats \ -in_wrapper ${pat_spec} Wgl The patterns specification is now updated with the new property setting.

b.Optionally, confirm the setting by inspecting the patterns specification:

SETUP>report_config_data $pat_spec

The patterns specification is displayed for examination.

8.Process the patterns specification with the process_patterns_specification command: SETUP> process_patterns_specification.

3.5.3 Generated MBIST verification patterns

A pattern specification for a MBIST-only design typically consists of:

One ICL verification pattern

One or multiple Memory BIST patterns

ICL verification patterns are automatically created from extracted ICL of the current design. These patterns ensure the ICL description is correct and matches the design's

implemented hardware. They check that all ICL-described Test Data Registers (TDRs) can be selected and have the expected length.

Memory BIST patterns exercise implemented MBIST controllers by clocking the design with appropriate clocks and instructing every BIST controller to launch a memory test. The generated testbenches (or patterns) thus run MBIST against design memories.

3.6 Memory BIST Diagnosis

Using memory BIST to test embedded memories provides significant advantage over the direct pin access test methods for PASS/FAIL testing. However, in most cases, it is important to identify the source of physical failure in the memory. This is referred to as the Diagnostic process.

Various diagnostic levels can be implemented on the chip depending on the diagnostic objectives. The following list provides various diagnostic levels and highlights the objectives of each level:

- Memory-Only Level
  - Identifies the failing memory only.
  - Is useful for small memories, where no root cause analysis is required.
- Memory Address Level
  - Identifies the failing address in a memory.
  - Provides limited failing address mapping.
  - Is useful for applications where soft repair based on address mapping is used.

- Real Time Bit Line Diagnosis
  - Identifies the failing column/row in a failing memory.
  - Is useful for offline repair.
- Offline Bitmapping
  - Mapping the failure to bits in memory.
  - Useful when detailed root cause analysis is to be performed for yield improvements.
- Online (Real Time) Bitmapping

o Maps the failure to bits in memory.

o Provides short diagnostic time; however, it requires significant tester memory and might require specific hardware to be implemented on the chip.

The most basic diagnostic feature enabled by memory BIST is shown in Figure 3.8. In this case, the memory BIST controller routes one or more compare status signals straight to chip pins. Cycle-by-cycle comparison findings are provided by these compare status signals. It is possible to set a comparison status output to represent anything from a single word in memory to an entire word. The tester can monitor each of these outputs since they are all directly wired to pins. The compare status diagnostics method is included into the memory BIST controller when the CompStat property of the ETPlanner configuration file is set to Yes or SharedWithGo.

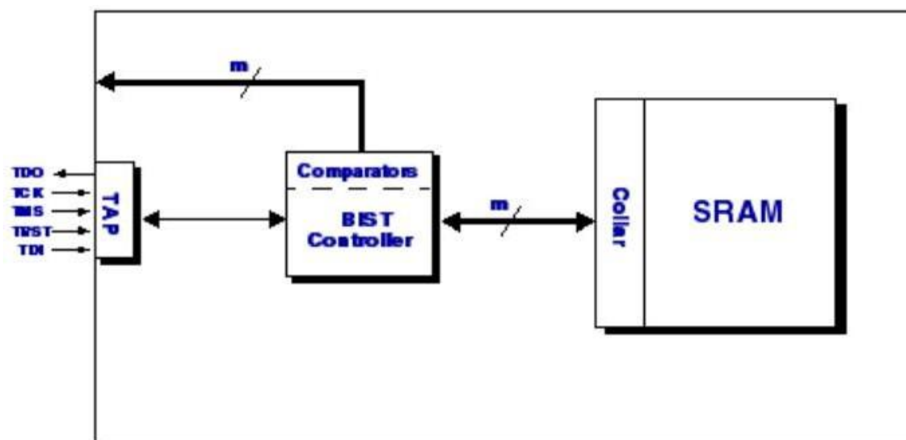

Fig. 3.8 Standard Diagnostic Approach Using Parallel Compare Status Lines [5]

This approach requires the tester to capture the Compare Status data at the same rate as the memory BIST controller run. It is common to run memory BIST on internally generated clocks which can be faster than the test clock. In this case, a purely scan-based diagnostic methodology also is supported, as shown in Figure. 3.9. Because the memory BIST controller has an error count register that is scan initialised before to each run, this technique is known as Stop-On-Nth- Error. The controller stops when it encounters several faults that match the number listed in the error register. This allows for the extraction of all relevant error data from the controller, including the algorithm step, address, failing bit position, and other details. By iteratively scanning in subsequent error counts and running the controller, all bit fail data can eventually be extracted. The Stop-On-Nth-Error

diagnostics option can be generated by memory BIST using the StopOnErrorLimit property of the ETPlanner configuration file.

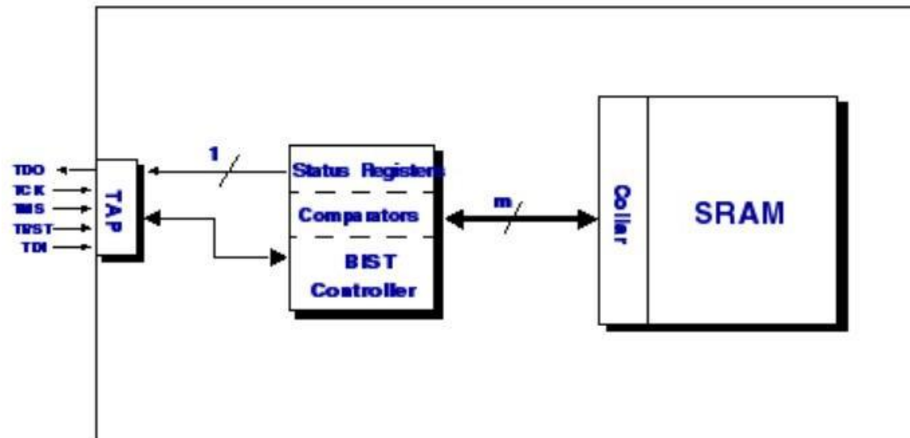

Fig. 3.9 Diagnostic Approach Using Stop-On-Nth-Error Serial Scan [5]

3.7 Implementation and Verification of Memory Repair

Repairable memories are popular in today's designs. However, the failure data collection, redundancy analysis, and memory repair access increase the design complexity. Furthermore, when a chip has multiple repairable memories, a solid infrastructure must be developed to integrate the repair analysis and fuse box programming at the chip level so that the memories can be repaired at chip power-up. Memory repair can be used to improve manufacturing yield of integrated circuits. Tessent MemoryBIST supports all types of memory repair: column repair, row repair, or a combination of both. Column repair includes replacing single columns or blocks of columns up to the full width of a column multiplexer. The full width case is often referred to as IO repair. Row repair includes replacing blocks, rows, or even a subset of a row, down to a single word.

Spare rows and columns are incorporated to the memory itself by the memory compiler. The memory repair characteristics must be described in the memory library file provided to the Tessent MemoryBIST tools.

Memory repair is performed in two steps. The first step consists of analyzing failures reported by the memory BIST controller during the test to determine if the memory is repairable and, if repairable, determining the values to be applied to the repair inputs of the memory.

The figure. 3.10 below shows the top-level architecture of the BISR chain and the BISR controller. A central fuse box is connected to a chip-level BISR controller. The central fuse box can be instantiated inside or outside the BISR controller module. When the fuse box is located outside the BISR controller module, extra connections between the fuse box and the BISR controller are required.

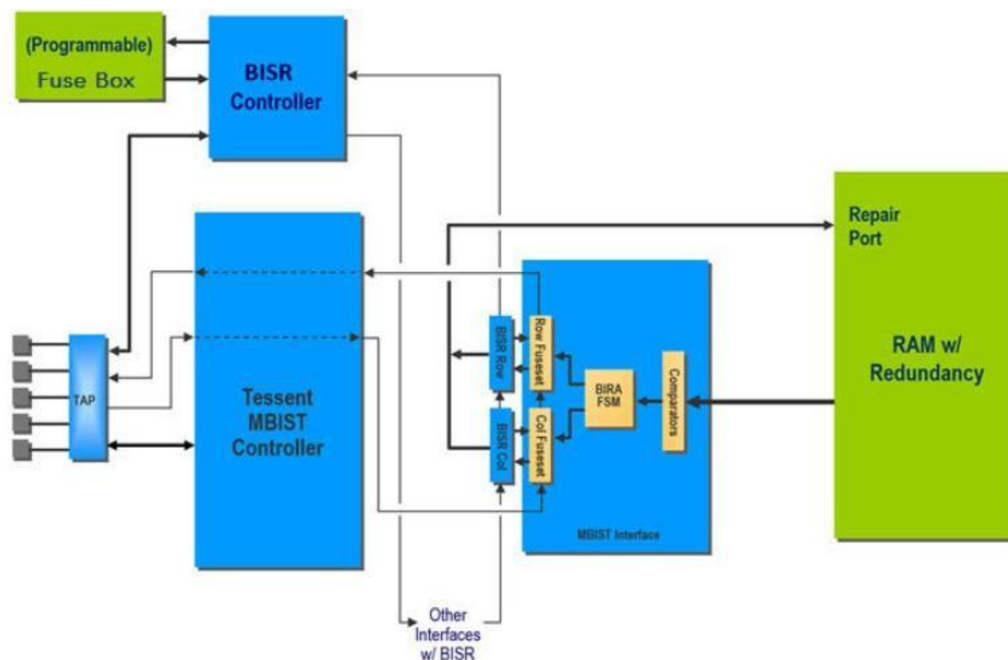

Fig. 3.10 Chip level repair architecture [6]

All repairable memories in the chip have a corresponding BISR register that holds a repair solution. All BISR registers in the chip are connected to form a chip-level BISR scan chain. The BISR chain is connected to a chip-level controller called a BISR controller. The BISR controller compresses the content of the BISR chain as it is scanned out of the BISR chain and writes the compressed data into the fuse box. The BISR controller can also decompress repair data from the fuse box and scan it in the BISR chain. The BISR controller works in conjunction with a BIRA

(Built-In Redundancy Analysis) module that provides the repair fuse values calculated from memory failure data.

Repairable memories implement spare or redundant rows or columns, enabling access to any faulty row or column to be redirected to a redundant element. The existing memory BIST diagnostic capabilities, such as the Compare Status pins diagnostic approach and the Stop-On- Nth-Error diagnostic approach, are not optimal solutions for the repair analysis of repairable memories. Each solution requires large test times or state-of-the-art high-speed memory testers. In addition, analysis of the results must be done afterwards to determine if a memory is repairable and what repair, if any, is required. To facilitate repairable memories in production testing environments, you can use the Built-In Repair Analysis (BIRA) feature to determine if a memory is repairable and the repair information based on the specified redundancy scheme. Memories with redundant IO as shown in Figure. 3.11 or Column elements as shown in Figure.

3.12 can be used to improve chip yield. IO repair replaces an entire memory sub-array and the associated columns for an IO, whereas column repair replaces a single column across one or more IOs. For IO replacement, the memory is repairable when one or more faults are along the same column, or one or more failing columns are within the same memory IO. In the column replacement mechanism, a single column is duplicated. The redundant element can repair one failing column across one or more memory IOs. For column replacement, the memory is repairable when one or more faults are along the same column or exactly one failing column is within the specified scope of repair.

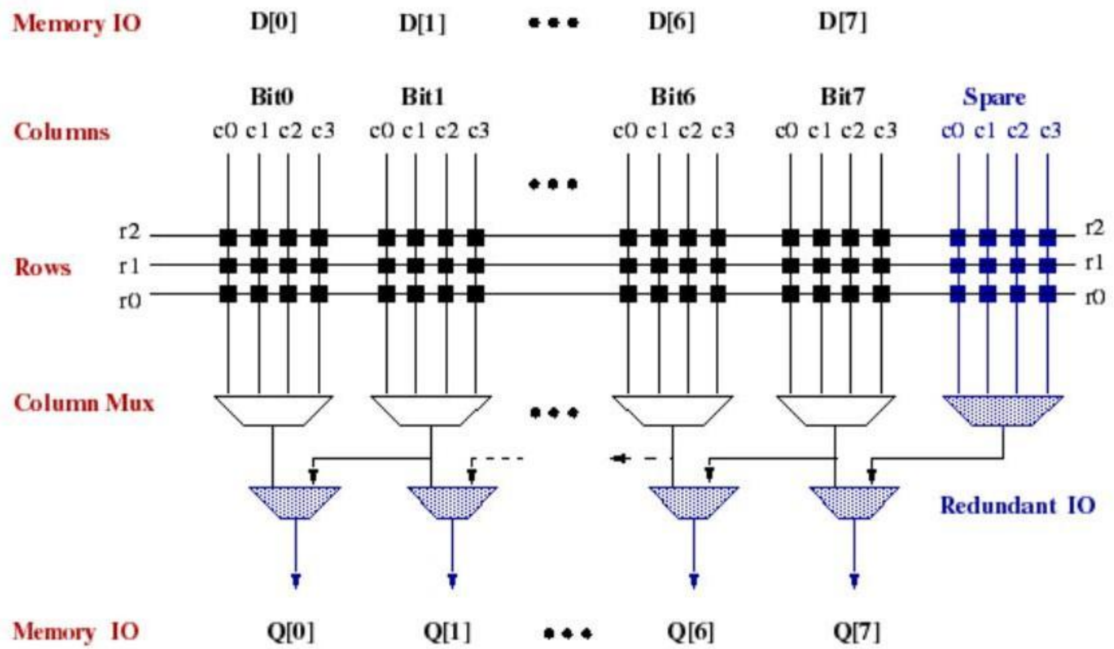

Fig. 3.11 Example of a memory with redundant IO [6]

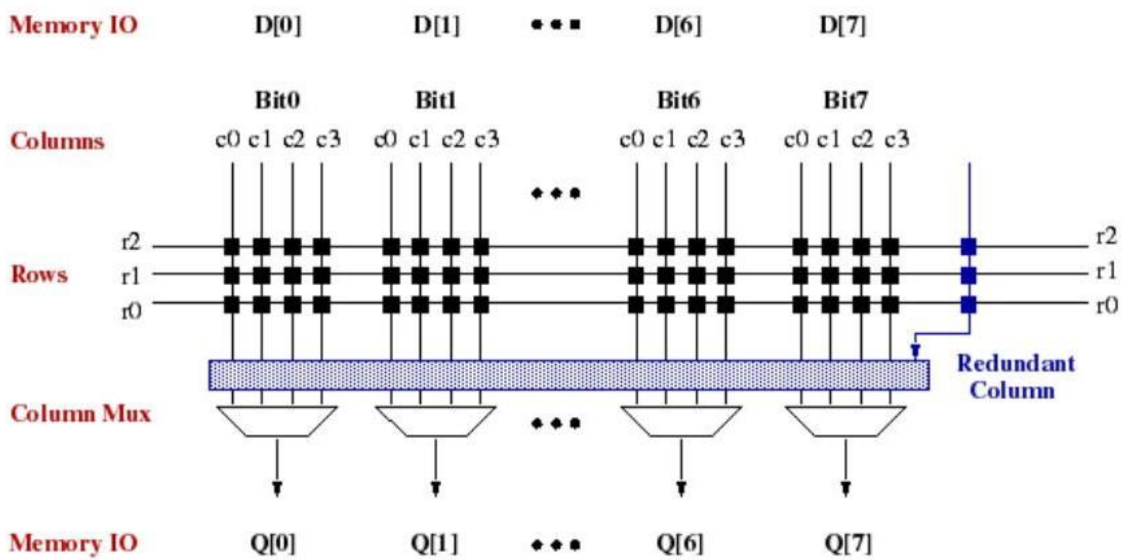

Fig. 3.12 Example of a memory with redundant column [6]

In the Row replacement mechanism, additional rows are built into the memory array. These redundant elements can repair any failing row within the entire memory or within a specific bank. The replacement mechanism depends on the memory design. For instance, a redundant element might replace a single physical row in one implementation, while another scheme always replaces multiple consecutive rows even if there is only one row failure.

3.8 Algorithms

Most typical memory errors have been found to be detectable using memory test techniques. Since many of these algorithms can service numerous on-chip memory and the hardware needed to generate the patterns is tiny, they are well suited for built-in self-testing (BIST). The March tests are the most widely used algorithms. When values are written to and read from known places in memory, March tests produce patterns that "march" up and down the memory addresses. The memory size and word length can be automatically calculated by these algorithms by retrieving the appropriate values from the memory model. March2 is the standard algorithm used to test RAMs. The default algorithm for testing ROMs is the rom1 (rom) algorithm. Key features of March tests include, Sequential Pattern Execution: The test reads and writes values in a sequential manner, moving up and down through memory addresses. This ensures that various types of memory access patterns are tested, increasing the likelihood of detecting faults. Bidirectional Testing: March tests go both up and down the address space, increasing test coverage and ensuring no memory locations are skipped. Automated Size and Word Length Detection: The algorithms can automatically calculate the memory size and word length by interacting with the memory model, eliminating the need for manual configuration. Common March algorithms are:

Step	Direction	Operation	Data/Transition
0	UP	W	0
1	UP	RW	0 -> 1
2	UP	RW	1 -> 0
3	DOWN	RW	0 -> 1
4	DOWN	RW	1 -> 0
5	DOWN	R	0

Table 3.1 Step Level Execution Sequence for March C

Step	Direction	Operation	Data/Transition
0	UP	W	0
1	UP	R	0
2	DOWN	W	1
3	DOWN	R	1

Table 3.2 Step Level Execution Sequence for March LR

Step	Direction	Operation	Data/Transition
0	UP	W	0
1	UP	RW	0 -> 1
2	UP	RW	1 -> 0
3	UP	RW	0 -> 1
4	DOWN	RW	1 -> 0
5	DOWN	RW	0 -> 1
6	DOWN	R	0

Table 3.3 Step Level Execution Sequence for March SS

Step	Direction	Operation	Data/Transition
0	UP	W	0
1	UP	RW	0 -> 1
2	DOWN	RW	1 -> 0
3	UP	RW	1 -> 0
4	DOWN	RW	0 -> 1
5	DOWN	R	0

Table 3.4 Step Level Execution Sequence for March Y

Step	Direction	Operation	Data/Transition
0	UP	W	0
1	UP	R	0
2	UP	R	0
3	DOWN	W	1
4	DOWN	R	1
5	DOWN	R	1

Table 3.5 Step Level Execution Sequence for March SR

ROM1 for ROMs: The rom1 (ROM) algorithm is typically used as the default testing algorithm for ROMs (Read-Only Memories). ROM testing algorithms are simpler than those for volatile memories because ROM data is non-volatile and not subject to frequent writes. ROM1 checks the integrity of stored data by reading each address and comparing it against expected values. Since ROMs cannot be written to in real-time during testing, the algorithm focuses on reading and verifying that no data corruption has occurred in storage. It is possible to implement additional algorithms that are not part of the Tessent library; these are typically called custom or User-Defined Algorithms (UDAs). These situations arise when, The memory architecture is unique and not fully testable by default algorithms. Specific fault models must be targeted that standard algorithms do not cover. Characteristics of UDAs: Custom Fault Models: UDAs can be tailored to detect specific

fault types, such as inter-cell faults, data retention faults, or timing-related issues that may not be adequately covered by traditional tests.

3.9 Methodological Framework

Now the objective of this work is to evaluate the contribution of individual test steps in these algorithms, as well as determining whether these test steps do not make any contribution to fault detection. By identifying these non-contributing test steps, it is possible to reduce MBIST test time without compromising fault coverage. The first step of the methodology is centered on modelling the structure of different March algorithms that have been widely used for MBIST test generation. In this paper, five different March algorithms have been considered, namely March C, March LR, March SS, March SR, and March Y. Each of these algorithms is represented as an ordered sequence of operations over all memory addresses. Each step of these algorithms is represented as a tuple of three parameters, namely address traversal direction, operation type, and data value or transition. The address traversal can be either UP (i.e., in ascending order) or DOWN (i.e., in descending order). The operation can be either write (denoted as W), read (denoted as R), or read-write (denoted as RW), while the data field can be either 0 or 1 or can denote a transition from 0 to 1 or 1 to 0.

The second part of the methodology is implementing an exhaustive fault injection model that will test the effectiveness of each step in identifying memory faults. The four fault models considered in this part of the methodology are stuck-at-0 (SA0), stuck-at-1 (SA1), transition faults (TF), and coupling faults (CF). These fault models generally identify common defects that can be found in memory circuits. In the simulation process, different memory addresses will have injected faults, and the expected value will be compared with the observed value in order to identify if there is a fault being detected. Write operations mainly initialize memory, while read/write operations identify faults during testing.

In order to avoid the dependency of the proposed redundancy analysis on a particular memory architecture, the experiments are performed over a variety of memory architectures. The memory depth varies from 256 to 4096 words, while the data width

varies from 8 to 256 bits. Additionally, the memory architectures used in the experiments include single-port memory architectures, dual-port memory architectures, memory architectures with ECC (Error Correcting Code) support, memory architectures without ECC support, memory architectures with BISR (Built-In Self-Repair) support, and memory architectures without BISR support. The selected March algorithms are then executed over the whole memory address space, and the detection behavior of each step of the March algorithms is recorded. After the completion of the simulation process, the fault detectability analysis of each step of the March algorithms is performed. Each step of the March algorithms is given a detection indicator depending on the detection of at least one fault. If a particular step of the March algorithms detects any fault during the simulation process, then the particular step of the March algorithms is said to be effective; otherwise, the particular step of the March algorithms is said to be redundant.

Redundancy analysis is then categorized into two types: local redundancy and global redundancy. A step is said to be locally redundant if it fails to detect any fault for a certain configuration. However, it may be able to detect faults for other configurations. A step is said to be globally redundant if it is redundant for most of the tested configurations. Only globally redundant steps are considered safe for removal since removing them will not impact the ability of the algorithm to detect faults.

Lastly, the potential test time reduction is estimated by comparing the number of steps remaining after the redundancy elimination process to the original number of steps in the algorithm. This is because each step involves traversing the memory array completely. Hence, the reduction of redundant steps will directly reduce the number of operations performed during the MBIST. This methodology thus provides a deterministic approach to the identification of redundant operations to optimize the test time of the MBIST algorithms without compromising the fault detection capability.

For the purpose of automating the evaluation process, a simulation framework has been implemented using Python. The simulation framework simulates the behaviour of the memory cells during the execution of the MBIST process and applies the operations specified by the March algorithm. During each step of the algorithm, the simulation framework traverses the memory addresses in the specified direction and applies the specified read, write, and read/write operations. During the execution of the simulation, the

framework keeps track of the number of cycles executed and the fault detection characteristics of each step of the algorithm. Detection vectors are generated to track the contribution of each step of the algorithm to the fault detection process. These vectors offer a systematic approach to the evaluation of each algorithm and the identification of the effectiveness of each step of the algorithm. By aggregating the results obtained from the execution of the simulation for multiple times, the patterns of redundancy in the algorithm structure are determined.

Once the detection data is accumulated, the results are processed to identify steps that do not contribute to fault detection. Steps where the result is zero for all the configurations are considered for removal. However, for safe optimization, only steps that show consistent redundancy for the majority of the configurations are considered for pruning. This ensures that the reliability of the MBIST process is not compromised. Finally, the redundant steps are used to estimate the improvement that can be achieved for the MBIST efficiency. Since each step of the March algorithm involves the traversal of the entire memory array, removing any single step can lead to substantial optimization of the operations performed during the test. This improvement in test cycles directly translates into improved efficiency, thereby saving time and resources during the manufacturing process.

CHAPTER-4

RESULTS AND ANALYSIS

After the MBIST insertion is completed successfully, it always produces three kinds of HDL output files BIST circuitry file, connection file and Testbench. BIST circuitry file is partition level design file with all the necessary BIST logic added, which further used for integration process by RTL teams. The connection file and the testbench are used to verify the functionality of the BIST circuitry. These files are described in greater detail in the following sections. Optionally, the tool can produce various other outputs: a pattern file, which contains either the input values from the BIST controller to the memory or the output values from the memory model, SDC file which will be delivered to backend team. The generated output files that use a specific set of naming conventions. The tool saves the generated output files with the default file name model_name_suffix.extension, where the model's HDL format determines the extension and the type of output determines the suffix. The BIST circuitry output file always includes the generated RTL for the BIST controller(s) specified in the dofile, and if in the insertion phase, it additionally includes the generated RTL of the memory collars for all relevant memory models that need collars. The BIST controller contains a finite state machine to control the operation of memory test algorithms that have selected. It contains the address generator, write data generator, expect data generator, and control signal generator. The controller usually contains a comparator to determine whether test results are good. The controller may also contain logic for optional features like diagnostics, BISA, and pipelining. The BIST collar instantiates the memory from original design in addition to test muxes and optional logic such as scan bypass logic and scrambling. The collar contains a compressor when not using comparators in the controller.

The connection file, named *<model_name>_bist_con.v* (or .vhd), and the test bench, named

<model_name>_tb.v (or .vhd), are used to validate the BIST circuitry. The connection file instantiates the BIST circuitry and connects all the ports together. The test bench instantiates the connection model and provides stimulus to start the BIST circuitry's algorithmic test activities and report test status when done. In the testbench, BIST is proceeding while test_h=1. The signal tst_done=1 indicates that BIST ran to completion.
Figure 4.1 gives a brief on overall project flow, left block represents the inputs required and right-side block represents the output that gets generated, the blocks in the centre represents the sequence of steps that are performed to obtain the result.

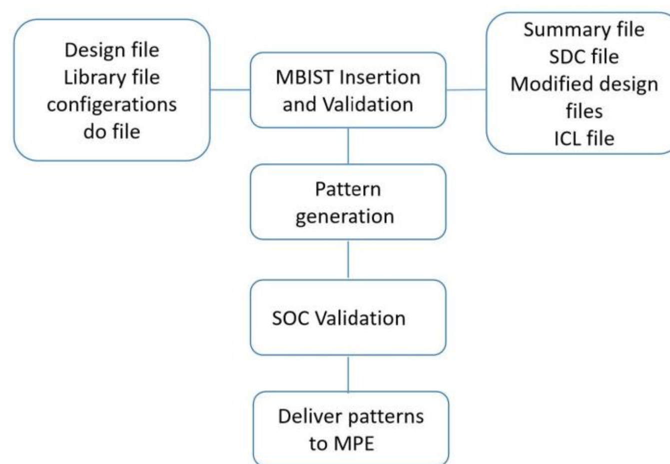

Figure. 4.1 Overall Project flow diagram

Figure. 4.2 represents the various output files that got generated after the MBIST insertion is completed. Design file top.v is the file that got generated post MBIST insertion which contains all the additional logic that got added while MBIST insertion. Similarly, icl, pdl and sdc files are also generated which will be used in further process.
Figure. 4.3 and Figure. 4.4 shows the design difference between the pre MBIST insertion and Post MBIST files.

```
ijtag_graybox/
modified_rtl_files/
top.design_source_dictionary
top.dft_info_dictionary
top.golden_design_source_dictionary
top.icl
top.ijtag_graybox_design_source_dictionary
top.pdl
top.sdc
top.tcd
top.tsdb_info
top.v_full@  --> modified_rtl_files/top.v
top.v_interface
```

Figure. 4.2 Files generated after MBIST Insertion

```
module top (clka, clkb, in1, out1, out2,tck,tdi,tdo,trst,tms,vddq,bisr_rstn, en_pd_A, en_pd_B,en_pd_C);

input clka, clkb, in1, vddq;
input bisr_rstn, en_pd_A, en_pd_B,en_pd_C;
output out1, out2;
input tck,tdi,trst,tms;
output tdo;

ipad clka_pad  (.PAD(clka), .C(clka_int));
ipad clkb_pad  (.PAD(clkb), .C(clkb_int));
ipad in1_pad ( .PAD (in1), .C(in1_int));
ipad bisr_rstn_pad (.PAD(bisr_rstn), .C() );
opad out1_pad ( .PAD(out1), .OEN(in1_int), .I(out1_int));
opad out2_pad ( .PAD(out2), .OEN(in1_int), .I(out2_int));
ipad tck_pad (.PAD(tck), .C());
ipad tdi_pad (.PAD(tdi), .C());
ipad trst_pad (.PAD(trst),.C());
ipad tms_pad (.PAD(tms),.C());
ipad vddq_pad  (.PAD(vddq), .C(vddq_int));
opad tdo_pad (.PAD(tdo), .OEN(1'b1), .I(1'b0));
ipad en_pd_A_pad (.PAD(en_pd_A), .C(en_pd_A_int));
ipad en_pd_B_pad (.PAD(en_pd_B), .C(en_pd_B_int));
ipad en_pd_C_pad (.PAD(en_pd_C), .C(en_pd_C_int));


cgand cgand1 ( .GCK(clka_gated) , .FE(out_int1) , .TE(1'b0) , .CK(clka_int) );
cgand cgand2 ( .GCK(clkb_gated) , .FE(out_int2) , .TE(1'b0) , .CK(clkb_int) );

core core_inst1 (.clka(clka_gated),
                 .clkb(clkb_gated),
                 .in(in1_int),
                 .out(out_int1));

core core_inst2 (.clka(clka_gated),
                 .clkb(clkb_gated),
                 .in(in1_int),
                 .out(out_int2));

my_fusebox_interface my_fusebox_interface (
   .clock(1'b0),
   .vddq(vddq),
   .address(10'b0),
   .write_en(1'b0),
   .select(1'b0),
   .access_en(1'b0),
   .scan_test(1'b0),
   .read_data(),
   .write_duration_count(32'b0),
   .done()
);
```

```
module top (clka, clkb, in1, out1, out2,tck,tdi,tdo,trst,tms,vddq,bisr_rstn, en_pd_A, en_pd_B,en_pd_C);

input clka, clkb, in1, vddq;
input bisr_rstn, en_pd_A, en_pd_B,en_pd_C;
output out1, out2;
input tck,tdi,trst,tms;
output tdo;

  wire [9:0] Address;
  wire top_rtl_tessent_tap_main_inst_so, top_rtl_tessent_tap_main_inst_tdo_en,
      tdi_pad_so, tms_pad_to_tms, top_rtl_tessent_sib_sri_inst_so,
      top_rtl_tessent_tdr_sti_ctrl_inst_so,
      top_rtl_tessent_tap_main_inst_to_select, top_rtl_tessent_sib_sti_inst_so,
      top_rtl_tessent_sib_pb1_inst_so, core_inst2_so,
      top_rtl_tessent_sib_sri_inst_to_select, top_rtl_tessent_sib_pb2_inst_so,
      core_inst1_so, top_rtl_tessent_sib_sti_inst_so_ts1,
      top_rtl_tessent_sib_sti_inst_to_select,
      top_rtl_tessent_sib_mbisr_inst_so,
      top_rtl_tessent_sib_pb1_inst_to_select,
      top_rtl_tessent_sib_pb2_inst_to_select, ijtag_to_reset, test_logic_reset,
      capture_dr_en, tck_pad_C, ijtag_to_se, ijtag_to_ce, shift_dr_en,
      ijtag_to_tck, ijtag_to_ue, update_dr_en, trst_pad_C, out_int1, out_int2,
      sti_ctrl_nonscan_test, sti_ctrl_nonscan_test_or2_Y,
      sti_ctrl_nonscan_test_or2_1_Y, host_bscan_to_sel, force_disable,
      select_jtag_input, select_jtag_output, scan_out, to_bscan_force_disable,
      to_bscan_select_jtag_input, to_bscan_select_jtag_output,
      to_bscan_capture_shift_clock, to_bscan_update_clock, to_bscan_shift_en,
      to_bscan_scan_in, from_bscan_scan_out, in1_int, EN1_en1, EN2_en1,
      in1_fromPad, out1_int, out1_toPad, out2_int, out2_toPad, vddq_int,
      vddq_fromPad, bisr_rstn_fromPad, en_pd_A_int, en_pd_A_fromPad,
      en_pd_B_int, en_pd_B_fromPad, en_pd_C_int, en_pd_C_fromPad, ltest_to_en,
      mbisr_clock, done, read_data, writeFB, selectFB, FBAccess,
      top_rtl_tessent_mbisr_controller_inst_SO, ijtag_to_sel, bisrSelM, bisrSE,
      bisrClk, bisrCEDis, bisrRstn, toBisr, bisrClk_pd_A, bisrRstn_pd_A,
      toBisr_pd_A, bisrClk_pd_B, bisrRstn_pd_B, toBisr_pd_B, bisrClk_pd_C,
      bisrRstn_pd_C, toBisr_pd_C, pd_A_bisr_so, pd_B_bisr_so, pd_C_bisr_so,
      bisr_so, pd_A_bisr_so_ts1, pd_B_bisr_so_ts1, pd_C_bisr_so_ts1,
      bisr_so_ts1;
  wire [31:0] strobeCnt;
ipad clka_pad  (.PAD(clka), .C(clka_int));
ipad clkb_pad  (.PAD(clkb), .C(clkb_int));
ipad in1_pad ( .PAD (in1), .C(in1_fromPad));
ipad bisr_rstn_pad (.PAD(bisr_rstn), .C(bisr_rstn_fromPad) );
opad out1_pad ( .PAD(out1), .OEN(EN1_en1), .I(out1_toPad));
opad out2_pad ( .PAD(out2), .OEN(EN2_en1), .I(out2_toPad));
ipad tck_pad (.PAD(tck), .C(tck_pad_C));
ipad tdi_pad (.PAD(tdi), .C(tdi_pad_so));
ipad trst_pad (.PAD(trst),.C(trst_pad_C));
ipad tms_pad (.PAD(tms),.C(tms_pad_to_tms));
ipad vddq_pad  (.PAD(vddq), .C(vddq_fromPad));
opad tdo_pad (.PAD(tdo), .OEN(top_rtl_tessent_tap_main_inst_tdo_en), .I(top_rtl_tessent_tap_main_inst_so));
```

Figure. 4.3 Additional logic that got added post MBIST insertion

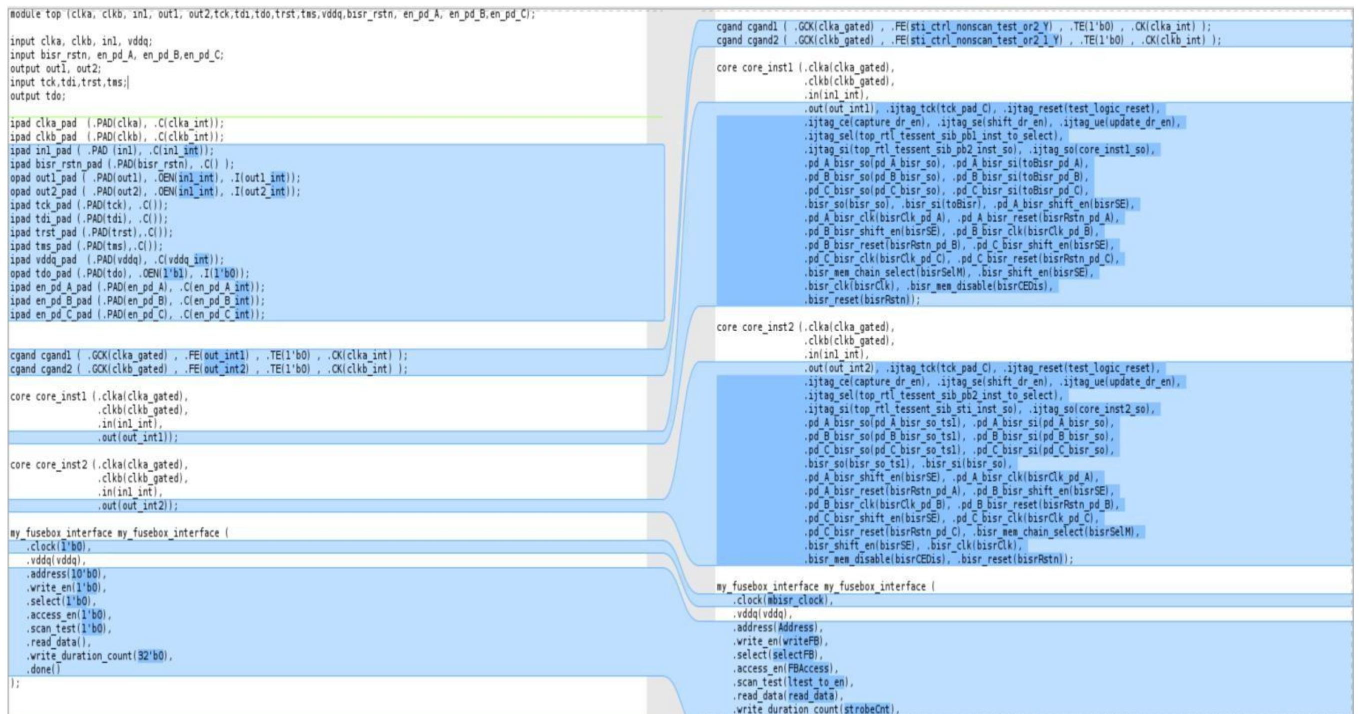

Figure. 4.4 Additional instances that got added post MBIST insertion

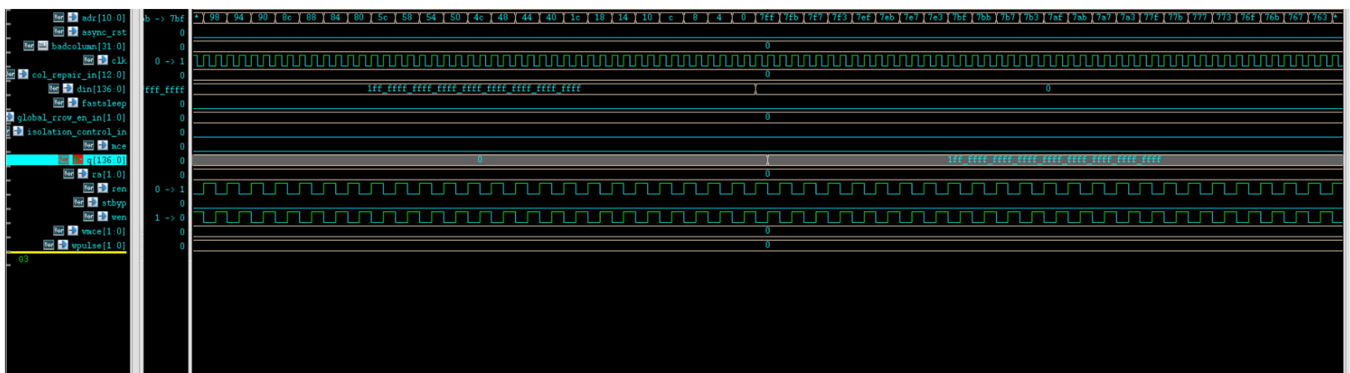

Fig 4.5 Waveform of SRAM ports in Synopsys Verdi

Figure. 4.6 shows the diagnostic simulation report, first the diagnostic algorithm is loaded, stop on nth error diagnosis is set and error counter is set to 0. Later the Done signal is checked before the BIST controller is enabled which result in fail. Now the BIST controller is enabled, and the Done signal is checked which is Pass. Based on the repair calculated by BIRA logic the address values, instruction pointer values and GO_ID values are modified in the PDL file. We can see that simulation is passing even when the fault is injected because of the modification in the PDL file the address of the faulty memory is directed to the redundant memory that is available.

```
13500ns:  Loading algorithm
13500ns:  Enabling diagnosis mode (GO_ID signals converted into compare status signals)
13500ns:  Enabling stop-on-nth-error diagnosis
13500ns:  Loading error counter with value 0
13500ns:  Enabling memory instance mem1 from memory test
143700ns:  Checking DONE is FAIL before execution
143700ns:  Starting MemoryBist controller execution : mbist_SRAM_c1_controller_inst
1.7595e+06ns:  Checking DONE is PASS after execution completion
1.7595e+06ns: Modified resource (tessent_mbist_bap_inst.mbist_done_0):
original Value (0b1): new value (0b0)
1.7672e+06ns:  Sampling registers containing diagnosis data
1.7672e+06ns:  Sampling 3-bit Operation Cycle Counter
1.7672e+06ns:  Sampling 5-bit Instruction Pointer
1.7672e+06ns:  Sampling 2-bit RepeatLoop A Counter
1.7672e+06ns:  Sampling 2-bit RepeatLoop B Counter
1.7672e+06ns:  Sampling 28-bit CounterA Counter
1.7672e+06ns:  Sampling 8-bit DelayCounter
1.7672e+06ns:  Sampling 9-bit X-Segment of Address Register A
1.7672e+06ns:  Sampling 9-bit X-Segment of Address Register B
1.7672e+06ns:  Sampling 2-bit Y-Segment of Address Register A
1.7672e+06ns:  Sampling 2-bit Y-Segment of Address Register B
1.7672e+06ns:  Sampling 4-bit Z-Segment of Address Register A
1.7672e+06ns:  Sampling 4-bit Z-Segment of Address Register B
1.7672e+06ns:  Sampling 16-bit Error Counter
1.7721e+06ns:  Modified resource (SRAM_c1_controller_inst.INST_POINTER_REG_HW[0:4]):
original Value (0b00000): new value (0b10100)
1.7721e+06ns: Modified resource (SRAM_c1_controller_inst.A_ADD_REG_X_HW[0:8]):
original Value (0b000000000): new value (0b000100000)
1.7721e+06ns:  Modified resource (GO_ID_REG[130:0]):
original Value
(0b00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000):
 new value
(0b00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000010)

Simulation finished at time 1861601
Number of miscompares        =      0
Number of 0/1 compares       =    979
Number of Z compares       =      0

No error between simulated and expected patterns
```

Fig. 4.6 Diagnostic pattern simulation report

The proposed methodology has been applied in the evaluation of the redundancy of test steps in various MBIST algorithms based on the March family. Deterministic simulation has been used in the analysis of the proposed methodologies. Five most commonly used March family-based MBIST algorithms have been evaluated in the proposed work. These are March C, March LR, March SS, March SR, and March Y. For each of the above-mentioned MBIST algorithms, the simulation of each and every step has been carried out across the entire memory address range. Various fault models, including stuck fault (SA0, SA1), transition fault (TF), and coupling fault (CF), have been used in the proposed work. A vector has been generated in the simulation framework for each of the above-mentioned MBIST algorithms. This vector will indicate whether the execution of a particular step is able to detect at least one fault or not. If no fault is detected by a particular step across all possible fault models, it is said that the execution of the particular step is redundant.

The results showed that some steps were redundant when they were examined locally for certain configurations. For example, the initialization step and the intermediate write step were not useful for fault detection in the March SR algorithm for certain configurations. In the March LR algorithm, two steps were also found to be locally redundant for certain configurations. However, not all locally redundant steps were safe to remove, as some of them were still useful for fault detection for certain configurations.

To remove redundant steps safely, a global redundancy check was also performed. In this case, the steps were considered redundant if they were redundant for at least 90% of the configurations. The results showed that only some of the locally redundant steps were safe to remove, as they did not meet the global redundancy criteria. It was observed that the initial write-only step (S0) for the March C algorithm and the March LR algorithm, along with one more step, met the global redundancy criteria. These steps were not able to detect faults for the majority of the configurations.

Based on these findings, the potential test-time reduction was estimated by comparing the number of retained steps with the total number of steps in each algorithm. The reduction percentage for each algorithm was calculated using the following formula:

$$\text{Reduction (\%)} = \frac{N - N_{retained}}{N} \times 100$$

where N represents the total number of steps in the original algorithm and $N_{retained}$ represents the number of steps remaining after removing globally redundant steps.

Applying this calculation to the analyzed algorithms produced the following reductions:

- **March C: T**otal steps = 6
  Globally redundant steps = 1
  Reduction = $\frac{1}{6} \times 100 = 16.67\%$
- **March LR:**
  Total steps = 4
  Globally redundant steps = 2
  Reduction = $\frac{2}{4} \times 100 = 50\%$
- **March SS:**
  No globally redundant steps identified
  Reduction = **0%**
- **March SR:**
  No globally redundant steps identified
  Reduction = **0%**
- **March Y:**
  No globally redundant steps identified
  Reduction = **0%**

To determine the overall improvement achieved by the proposed framework, the reduction values across all evaluated algorithms were averaged. The average reduction is computed as:

$$\text{Average Reduction} = \frac{16.67 + 50 + 0 + 0 + 0}{5}$$

$$\text{Average Reduction} = 13.33\%$$

Thus, the proposed step-level redundancy analysis achieves an average MBIST test-time reduction of approximately 13.33% across the evaluated algorithms without compromising fault coverage.

These results demonstrate that although redundancy exists within several March algorithms, only a limited number of steps are consistently redundant across configurations. The analysis confirms that removing such globally redundant steps can provide measurable improvements in MBIST efficiency while preserving the reliability of memory testing.

Algorithm	Total Steps	ocally Redundant Steps	Local Reduction(%)
March C	6	0	16.67
March LR	4	0, 2	50.00
March SS	7	0	14.29
March SR	6	0, 3	33.33
March Y	6	0	16.67

Table 4.1. Local Step Redundancy of all March Algorithms

lgorithm	Total Steps	ocally Redundant Steps	Global Reduction(%)
March C	6	0	16.67
March LR	4	0, 2	50.00
March SS	7	-	0.00
March SR	6	-	0.00
March Y	6	-	0.00

Table 4.2. Global Step Redundancy of all March Algorithms

The new optimized version of the MBIST algorithm was then analyzed under different fault models such as stuck-at faults (SA), transition faults (TF), and coupling faults (CF) through the use of the aforementioned Python based 2-Dimensional memory simulation framework. The conventional March sequences were used as a basis, and an analysis on the baseline performance in terms of execution time and fault coverage was conducted first. Then, the optimized algorithm was implemented after removing the redundant operations using the redundancy pruning technique.

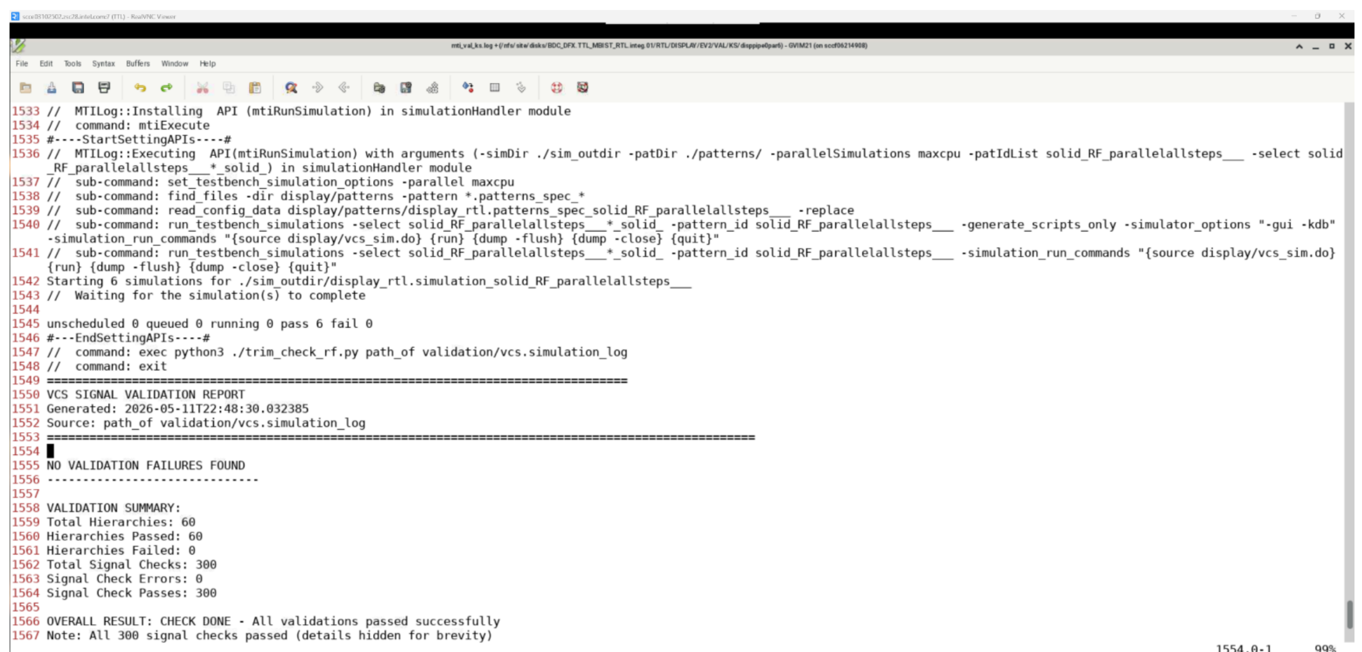

Fig 4.7 Simulation log before optimization of Algorithms

```
630197 1.268e+07ps: Checking GO is PASS after execution completion
630198 1.268e+07ps: Checking DONE is PASS after execution completion
630199 1.506e+07ps: Checking GO is PASS after execution completion
630200 1.506e+07ps: Checking DONE is PASS after execution completion
630201 Simulated 1 scan patterns
630202   Scan pattern range: 0 to 0
630203 Patterns passed: 1
630204 Patterns failed: 0
630205
630206 Simulation finished at time 55530001
630207 Number of miscompares           =          0
630208 Number of 0/1 compares           =       3559
630209 Number of Z compares             =          0
630210 █
630211 No error between simulated and expected patterns
630212
630213 $finish called from file "./patterns_directory/PMOVIFastx_solid_.sv", line 18001.
630214 $finish at simulation time 55530002.000ps
630215            V C S   S i m u l a t i o n   R e p o r t
630216 Time: 55530002000 fs
630217 CPU Time:     122.201.820 seconds;       Data structure size:  15.2Mb
630218 Mon May 12 12:33:20 2026
630219 CPU time: 106.007 seconds to compile + 1.322 seconds to elab + .994 seconds to link + 13.878 seconds in simulation
630220 if [ $? != 0 ] ; then xitval=-1; fi
630221
630222 if [ $xitval -ne 0 ]; then
630223   echo "Simulation failed"
630224 fi
630225
630226 exit $xitval
630227 exit0
```

Fig 4.8 Simulation log after optimization of Algorithms

From the results of the experimental analysis, it can be observed that the new optimized algorithm was able to provide shorter execution time than the conventional March sequences. Since the new algorithm did not contain any unnecessary read and write operations, it performed faster without sacrificing stable testing behavior especially for simpler faults such as stuck-at faults and transition faults. The reduction in time was greater in large memory sizes since March operations run through all the memory cells. Moreover, unlike stochastic methods like the one using genetic algorithm for sequence generation, the new algorithm was more predictable in nature and scalable in performance.

CHAPTER-5

CONCLUSION, FUTURE AND SOCIAL IMPACT

5.1 Conclusion

This work presented a deterministic framework for identifying step-level redundancy in widely used March-based Memory Built-In Self-Test (MBIST) algorithms. By modeling the internal structure of March algorithms and performing exhaustive fault injection across multiple memory configurations, the study evaluated whether individual test steps contribute to fault detection. The methodology incorporated common memory fault models including stuck-at, transition, and coupling faults, ensuring that the redundancy analysis was conducted under realistic worst-case testing conditions.

The experimental analysis revealed that certain steps within March algorithms do not contribute to fault detection under many configurations. These steps were first identified as locally redundant through step-level detection analysis and then evaluated for consistency across configurations. Only those steps that remained redundant in the majority of configurations were classified as globally redundant and considered safe for removal. This conservative approach ensured that the optimization process did not compromise the fault coverage of the MBIST algorithms.

The results showed that globally redundant steps exist in some commonly used March algorithms, particularly in the initialization stages. By removing these steps, it was possible to reduce the number of memory operations executed during testing. The calculated results demonstrated an average MBIST test-time reduction of approximately 13.33% across the evaluated algorithms while maintaining the same level of fault detection capability.

Overall, the proposed methodology provides a systematic and deterministic approach for analyzing redundancy within MBIST algorithms. The framework establishes a safe lower bound for test-time optimization and highlights the potential for improving testing efficiency in memory-intensive System-on-Chip designs. Future work may extend this approach by incorporating adaptive techniques or machine learning models to dynamically optimize MBIST execution for specific memory architectures and fault distributions.
Only those steps which were redundant in the majority of the configurations were classified as globally redundant and were considered to be safe for removal. Such an approach ensured that the optimization process did not compromise the fault coverage of the MBIST algorithms.

The results indicated the existence of globally redundant steps in some of the popularly used March algorithms, specifically during the initialization steps. Removal of such steps allowed for the reduction of the number of memory operations during the testing phase. The calculated results indicated the potential for reducing the test time by approximately 13.33% for the evaluated algorithms, with the same level of fault detection capability.

The proposed methodology for the analysis of the redundancy of the MBIST algorithm has been able to offer a systematic and deterministic approach for the optimization of the test time for the System-on-Chip designs. Future studies could focus on the integration of the proposed methodology with some adaptive approach to optimize the execution of the MBIST algorithm for specific memory architectures.

5.2 Future Scope

The future of Memory Built-In Self-Test (MBIST) is promising as semiconductor technology advances. As integrated circuits (ICs) continue to evolve in terms of complexity, size, and functionality, the role of MBIST will become increasingly critical. Here are several key areas where MBIST is expected to see significant advancements and impact: With advancements in nanotechnology, we could see the development of nanoscale repair agents capable of interacting with memory structures at a granular level. These agents might autonomously identify defects and repair memory cells at the atomic scale, leading to more durable and self-healing memory systems. As memories become more

complex, error correction mechanisms that work in conjunction with MBIST could improve. Advanced ECC (Error Correction Codes) could be integrated within MBIST to not only detect errors but also correct them in real time, ensuring higher reliability in mission-critical applications like AI, automotive, and aerospace.

Future MBIST systems may become more adaptive, capable of dynamically generating and optimizing test patterns based on real-time analysis of the IC's behavior and fault statistics. This will significantly improve the effectiveness of generated test patterns, resulting in reduced test time while maintaining high fault coverage. Leveraging machine learning algorithms to analyze historical fault data and predict the most effective test patterns could reduce test iterations and improve the efficiency of MBIST, minimizing test time while maximizing coverage. Combining functional and structural test techniques within MBIST frameworks could ensure that even the most complex memory architectures are thoroughly validated, addressing both memory cell defects and more intricate timing or logic faults.

As the industry moves towards advanced semiconductor nodes (such as sub-5nm processes), MBIST will need to adapt to new memory technologies like 3D NAND, MRAM, ReRAM, and PRAM. These emerging memory types will present new fault models and require specialized algorithms for effective testing and repair. With the rise of 3D ICs and chiplet architectures, MBIST systems will need to evolve to support memory testing across multiple stacked dies and chiplets. This will require more sophisticated interconnect testing and the ability to operate efficiently in complex multi-layered environments.

MBIST could evolve to include real-time monitoring capabilities that continuously diagnose memory health during the operation of the IC. This would enable proactive fault detection and repair, ensuring that systems remain reliable even in mission-critical applications.

Test Time Minimization: As ICs become more complex, minimizing the time required to test memory components is critical. Faster pattern generation and more efficient memory access methods will help reduce test times. This is especially important in the era of high-performance computing (HPC) and data center processors, where memory sizes are massive, and testing can become a bottleneck. Power-Aware Testing: Future MBIST systems may incorporate power- aware algorithms to ensure that the test process itself does

not consume excessive power, especially in low-power applications like mobile devices or IoT sensors. Optimizing test patterns for power efficiency will be a key area of focus. MBIST in Autonomous Vehicles: With the rise of autonomous vehicles, the reliability of on-chip memory becomes even more critical. In the future, MBIST could evolve to provide continuous, in-field testing of memory components in automotive applications, ensuring system safety without requiring the vehicle to go offline for maintenance. IoT and Edge Computing: In IoT devices and edge computing nodes, where devices are often deployed in remote or harsh environments, MBIST will play a crucial role in ensuring long-term reliability. Self-test and self- repair mechanisms embedded within these devices will ensure they can operate without human intervention for extended periods.

As MBIST technologies advance, there will likely be a push for greater standardization to ensure interoperability across different vendors and tools. This would make it easier to implement MBIST solutions in a wide variety of IC designs without needing to customize them for each specific use case.

5.3 Social Impact

While Memory Built-In Self-Test (MBIST) is primarily a testing technique used to verify the reliability and functionality of memory components in integrated circuits (ICs), its implementation does have some environmental implications. These impacts stem from the resources and energy consumption involved in the design, manufacturing, and testing phases of semiconductor production. Here's a detailed breakdown of the environmental impacts associated with MBIST, Energy Consumption During Manufacturing: MBIST involves embedding test logic within memory components, which adds complexity to the IC design. This increased complexity can require more design resources, higher computational power, and longer design cycles. In turn, this can lead to higher energy consumption during the design and manufacturing processes, particularly if design tools and foundry equipment are not optimized for energy efficiency. Material and Resource Usage: The design and testing processes for MBIST may also require the use of additional materials, such as photomasks, during the IC fabrication process. Depending on the volume and intricacy of the IC design, this can contribute to the use of raw materials and potentially increase waste generation if the process is not managed efficiently.

Testing Equipment Energy Usage: The testing phase in semiconductor manufacturing, where MBIST verifies the functionality of memory components, requires specialized testers and handlers. These machines can consume significant amounts of energy, particularly in large-scale production environments where hundreds or thousands of ICs are tested simultaneously. While the energy consumed by MBIST itself is relatively small compared to other manufacturing steps, the cumulative impact can become significant in high-volume production. Indirect Impact on Facility Energy Consumption: Semiconductor manufacturing facilities, known as fabs, are energy-intensive operations. The additional steps required for MBIST testing may contribute to overall facility energy consumption. If the testing processes are not optimized for efficiency, this could lead to higher emissions of greenhouse gases (GHG), especially if the fab relies on non-renewable energy sources.

Discarded Defective Components: One of the functions of MBIST is to identify faulty memory components during the manufacturing process. While this helps ensure that only fully functional ICs are shipped, it can also lead to the generation of waste from defective components. If these

components are not managed or recycled properly, they may contribute to e-waste, which poses significant environmental hazards due to the presence of toxic materials such as heavy metals and hazardous chemicals in ICs. Chemical Waste from Testing Procedures: During MBIST testing, some chemical waste may be generated, especially in relation to testing procedures that involve specialized chemicals for verification or defect analysis. If not handled properly, this can contribute to environmental pollution.

Improved Reliability and Product Longevity: Despite the potential environmental costs associated with its implementation, MBIST plays a vital role in improving the reliability and longevity of memory components in electronic devices. By ensuring that memory components function correctly, MBIST helps reduce the likelihood of device failure, which can extend the lifespan of electronic products. This has significant environmental benefits by reducing the need for frequent replacements, leading to a decrease in e-waste over time. Reduction in Manufacturing Defects: MBIST allows for the early identification of defective memory components during the production phase. This preemptive detection helps manufacturers avoid shipping faulty products, reducing the

need for recalls and replacements that would otherwise generate unnecessary waste and energy consumption in downstream processes.
Indirect Contribution to Energy-Efficient Devices: MBIST testing ensures that memory components are reliable and power-efficient, indirectly contributing to the overall energy efficiency of electronic devices. When memory components operate more efficiently, they consume less power during the use phase, which helps reduce the energy footprint of devices like smartphones, computers, and servers. This is particularly important as data centers and edge devices demand increasingly high levels of efficiency to minimize their environmental impact. Optimizing Test Patterns for Efficiency: As MBIST evolves, improvements in test pattern generation can help optimize the testing process, reducing the time and energy required for testing. Efficient testing processes result in lower energy consumption and reduced emissions during the manufacturing phase. This optimization is particularly important for high-volume production environments where even small improvements in test efficiency can lead to significant reductions in environmental impact.

References

[1] Abramovici, M., Breuer, M. A., & Friedman, A. D. (1990). Digital Systems Testing and Testable Design. IEEE Press

[2] Agarwal, V., & Seth, S. C. (1989). "Test Generation for VLSI Chips." IEEE Design & Test of Computers, 6(6), 14-25.

[3] K. R. Raval, Y. D. Parmar and C. R. Panchal, "DFT methodology for memory testing on lower technological node," 2017 International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2017, pp. 430-435, doi: 10.1109/ICCMC.2017.8282725.

[4] P. Arora, P. Gallagher and S. L. Gregor, "Core Test Language based High Quality Memory Testing and Repair Methodology," 2021 IEEE International Test Conference India (ITC India), Bangalore, India, 2021, pp. 1-6, doi: 10.1109/ITCIndia52672.2021.9532722.

[5] Tessent® MemoryBIST User's and Reference Manual

[6] Tessent® IJTAG user's manual

[7] VCS/VCSI User Guide

[8] J. Rearick, B. Eklow, K. Posse, A. Crouch and B. Bennetts, "IJTAG (internal JTAG): a step toward a DFT standard," IEEE International Conference on Test, 2005., Austin, TX, USA, 2005, pp. 8 pp.-815, doi: 10.1109/TEST.2005.1584044. keywords: {System testing;Memory management;Built-in self-test;Circuit testing;Logic testing;Pins;Access protocols;Integrated circuit interconnections;Clocks;Printed circuits},

[9] S. Zhang, Z. Wang and Y. Wang, "The MBIST Implementation of SoC Chip," 2024 9th International Conference on Integrated Circuits and Microsystems (ICICM), Wuhan, China, 2024, pp. 73-77, doi: 10.1109/ICICM63644.2024.10814273. keywords: {Micromechanical

devices;Memory management;Random access memory;Manufacturing;System-on-chip;Logic;Chip scale packaging;Phase locked loops;Standards;Testing;radiation;SoC;DFT;MBIST;JTAG;TAP;at-speed},

[10] R. Yang, Z. Wang and M. Shen, "An Efficient Grouping Method for Large-Scale MBIST," 2024 2nd International Symposium of Electronics Design Automation (ISEDA), Xi'an, China, 2024, pp. 486-491, doi: 10.1109/ISEDA62518.2024.10617536. keywords: {Greedy algorithms;Design automation;Annealing;Discrete Fourier transforms;Simulated annealing;Built-in self-test;Design for testability;MBIST grouping;greedy algorithm;simulated annealing algorithm},

[11] S. K B and B. K R, "Strategic DFT Implementation and Verification of Next–Gen Electronics Using ATPG Simulation," 2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS), Bengaluru, India, 2024, pp. 1-6, doi: 10.1109/CSITSS64042.2024.10817045. keywords: {Computational modeling;Discrete Fourier transforms;Built-in self-test;Reliability engineering;Vectors;Manufacturing;Delays;Circuit faults;Test pattern generators;Integrated circuit modeling;ATPG (Automatic test pattern generator);DFT (Design for testability);Fault Coverage;Stuck-At fault;Transition fault},

LIST OF PUBLICATION AND PROOF

1. **Kaustubh Ranjan, Sahil Srivastava** "Optimization of MBIST March Algorithms Using Structural Redundancy Pruning and Machine Learning Guided Configuration Selection" . Submitted. Unpublished in the 6th International Conference on Emerging VLSI & Semiconductor Technology for AI and Computing Applications (EVST–2026)

Fwd: Acceptance Notification of Paper ID # 614 | EVST-2026 | IEEE Conference Record No. 69093 | KIET (Deemed to be University), Delhi NCR, Ghaziabad

External Inbox ×

Convert to PDF

Kaustubh Ranjan Singh <kaustubhr.singh@gmail.com>

to me

Thu, Mar 12, 12:23 PM

Thanks and Regards,
Kaustubh Ranjan Singh

---------- Forwarded message ---------
From: **Microsoft CMT** <noreply@msr-cmt.org>
Date: Thu, 12 Mar 2026, 10:52
Subject: Acceptance Notification of Paper ID # 614 | EVST-2026 | IEEE Conference Record No. 69093 | KIET (Deemed to be University), Delhi NCR, Ghaziabad
To: Kaustubh Ranjan Singh <kaustubhr.singh@gmail.com>

Dear Kaustubh Ranjan Singh,

We are pleased to inform you that your research paper titled "Optimization of MBIST March Algorithms Using Structural Redundancy Pruning and Machine Learning Guided Configuration Selection" (Paper ID: #614) has been conditionally accepted for presentation at the 6th International Conference on Emerging VLSI and Semiconductor Technology for AI and Computing Applications 2026 (EVST- 2026) [Conference Record #69093] organized by the Department of Electronics and Communication Engineering, KIET (Deemed to be University), Delhi NCR, Ghaziabad, and scheduled to be held on 26th -27th June 2026.

You are kindly requested to complete the registration process for your accepted paper by remitting the applicable conference registration fee. Details regarding the registration fee structure and applicable categories are available at the following link:
https://www.old.kiet.edu/evst2026/index.php?%2Fhome%2Fsubmission#registrationfee

You may complete your registration using the link provided below: https://tech.kiet.edu/kiet-epay/eventName/?id=67a9c862-ec8e-4e7f-867f-356681326a1f

Please note that the registration must be completed on or before 7 days of this email. Failure to complete the registration within the stipulated timeframe will result in the automatic cancellation of the paper's acceptance.

Please also note that the registration fee covers a standard paper length of up to six (6) pages. If the camera-ready version of your research paper exceeds six pages, an additional charge of ₹1000 (for Indian authors) or USD 15 (for international authors) per extra page will be applicable as specified on the conference website: https://www.old.kiet.edu/evst2026/index.php?%2Fhome%2Fsubmission#registrationfee. These additional charges are to be paid during the subsequent stages of the submission process.

Further instructions and details will be communicated after the successful completion of the registration process.

In case of any difficulty related to fee payment, please feel free to contact Mr Mohit Tyagi (Assistant Professor) at +91-9643142808 for assistance.

Thank you for choosing EVST-2026 as a platform to present and publish your research work.

Thanks & regards,
Organizing Chair
EVST-2026

Thesis Content Part.pdf

ORIGINALITY REPORT

9%	7%	2%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

#	Source	Type	%
1	www.asset-intertech.com	Internet Source	5%
2	manualzz.com	Internet Source	1%
3	"Design for AT-Speed Test, Diagnosis and Measurement", Springer Nature, 2002	Publication	1%
4	www.coursehero.com	Internet Source	1%
5	www.besttest.com	Internet Source	<1%
6	www.scribd.com	Internet Source	<1%
7	Shamik Das, An Chen, Matt Marinella. "Beyond CMOS", 2021 IEEE International Roadmap for Devices and Systems Outbriefs, 2021	Publication	<1%
8	Logic Synthesis Using Synopsys®, 1995.	Publication	<1%
9	S. Pateras. "IP for embedded diagnosis", IEEE Design & Test of Computers, 2002	Publication	<1%

Exclude quotes	On	Exclude matches	< 3 words
Exclude bibliography	On		

SAHIL SRIVASTAVA

Phone : +91 7827469932
Email - sahil.srivastava.3501@gmail.com
LinkedIn - https://www.linkedin.com/in/sahil-srivastava-9767851b3/

EDUCATION

M.TECH (VLS)	2024-2026	Delhi Technological University, Rohini, New Delhi -110042	NA
B. TECH (ECE)	2019-2023	Mahavir Swami Institute of Technology, Sonipat, Haryana	87.3%
CBSE (Class XII)	2018	Gandhi Memorial Government Boys Senior Secondary School, Shahdara, Delhi	86.4%
CBSE (Class X)	2016	Flora Dale School, Dilshad Garden, Delhi	70.3%

OBJECTIVE

Aspiring semiconductor engineer actively working in MBIST development for next-generation SoCs, including automated flow creation, controller integration, and multi-level validation in one of the most reputed semiconductor company in the world. Seeking a role that allows me to apply strong fundamentals in digital design, scripting, and DFT to deliver robust, high-coverage test solutions in cutting-edge chip design projects.

Technical SKILLS

Technical Courses	Tools	Languages
Technical Courses: Digital Electronics, Digital IC Design, Linux Scripting, Digital Design using Verilog, Low power VLSI techniques, Physical Design Flow, Static Timing Analysis, RTL to GDS flow	**Tools**: Cadence Virtuoso, Xilinx ISE, PROTEUS, LT Spice, Synopsys Verdi, Siemens Tessent	**Languages**: Verilog

Work Experience

Graduate Technical Intern, Intel Corporation — **June 2025 – Present**

Executed MBIST insertion and integration for next-generation SoCs, ensuring seamless connectivity between BIST controllers, memory instances, and top-level interfaces.
Performed block-level and hub-level MBIST validation, identifying design issues early and improving overall test coverage and reliability.
Developed and optimized automation scripts Python to streamline the complete MBIST flow, reducing manual effort and improving turnaround time.
Debugged MBIST insertion issues using simulation logs, waveforms, and tool reports, ensuring correct address/data bus mapping, repair logic, and test algorithm execution.
Gained hands-on experience with March algorithms, memory repair flows, test pattern generation, and debug in a production-grade DFT environment.

Subject Matter Expert, Coursehero — **Apr 2023 – Aug 2024**

Catered to the doubts of students from the engineering field and helped them in various subjects like digital electronics, control system , engineering mathematics, etc.

Subject Matter Expert, Chegg Inc — **Jan 2022 – Dec 2022**

Maintained a CF Score of 85% and above.
Worked on various software like LT Spice , MATLAB to tackle the problems raised by students.

ACADEMIC PROJECTS

Layout of 4bit Carry Ahead Adder

Designed layout for a 3x8 decoder and performed successful DRC and LVS checks and then proceeded to pre layout vs post layout simulations after RCX extraction.
Technology: Cadence Virtuoso (gpdk 90)

Implementation of Multi-Bit Latch using Low power techniques

Designed a 4-bit latch using pulse generator as clock which uses stacked inverters. Performing DRC and LVS checks and then will proceed to pre layout vs post layout simulations after RCX extraction.
Technology: Cadence Virtuoso (gpdk 90)

POSITIONS OF RESPONSIBILITY

Teaching assistantship under Mr. Kaustubh Ranjan Singh — **Aug 2024 – Apr 2025**

ACHIEVEMENTS

Qualified GATE 2023 and GATE 2024.
GOLD Medalist in Inter State Skating Competition