IMPLEMENTATION OF MEMORY BIST CONTROLLER IN FPGA USING THE IMPROVED MARCH AZ1 AS THE TEST ALGORITHM

CLEMENT OOI JUN JIE

UNIVERSITI TEKNIKAL MALAYSIA MELAKA

IMPLEMENTATION OF MEMORY BIST CONTROLLER IN FPGA USING THE IMPROVED MARCH AZ1 AS THE TEST ALGORITHM

CLEMENT OOI JUN JIE

This report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Electronics Engineering Technology with Honours

Faculty of Electronics and Computer Technology and Engineering
Universiti Teknikal Malaysia Melaka

2025

UNIVERSITI TEKNIKAL MALAYSIA MELAKA
FAKULTI TEKNOLOGI DAN KEJURUTERAAN ELEKTRONIK DAN KOMPUTER

BORANG PENGESAHAN STATUS LAPORAN
PROJEK SARJANA MUDA II

Tajuk Projek	:	**IMPLEMENTATION OF MEMORY BIST CONTROLLER IN FPGA USING THE IMPROVED MARCH AZ1 AS THE TEST ALGORITHM**
Sesi Pengajian	:	2024/2025

Saya CLEMENT OOI JUN JIE mengaku membenarkan laporan Projek Sarjana Muda ini disimpan di Perpustakaan dengan syarat-syarat kegunaan seperti berikut:

1. Laporan adalah hakmilik Universiti Teknikal Malaysia Melaka.
2. Perpustakaan dibenarkan membuat salinan untuk tujuan pengajian sahaja.
3. Perpustakaan dibenarkan membuat salinan laporan ini sebagai bahan pertukaran antara institusi pengajian tinggi.
4. Sila tandakan ( ):

☐	**SULIT***	(Mengandungi maklumat yang berdarjah keselamatan atau kepentingan Malaysia seperti yang termaktub di dalam AKTA RAHSIA RASMI 1972)
☐	**TERHAD***	(Mengandungi maklumat terhad yang telah ditentukan oleh organisasi/badan di mana penyelidikan dijalankan.
✓	**TIDAK TERHAD**	

Disahkan oleh:

Alamat Tetap:

DR. AIMAN ZAKWAN BIN JIDIN
Pensyarah Kanan
Fakulti Teknologi dan Kejuruteraan
Elektronik dan Komputer
Universiti Teknikal Malaysia Melaka

Tarikh : 12 Februari 2025

Tarikh : 14 Februari 2025

*CATATAN: Jika laporan ini SULIT atau TERHAD, sila lampirkan surat daripada pihak berkuasa/organisasi berkenaan dengan menyatakan sekali tempoh laporan ini perlu dikelaskan sebagai SULIT atau TERHAD.

DECLARATION

I declare that this project report entitled "**IMPLEMENTATION OF MEMORY BIST CONTROLLER IN FPGA USING THE IMPROVED MARCH AZ1 AS THE TEST ALGORITHM**" is the result of my own research except as cited in the references. The project report has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature	:	
Student Name	:	CLEMENT OOI JUN JIE
Date	:	31/12/2024

APPROVAL

I hereby declare that I have checked this project report and in my opinion, this project report is adequate in terms of scope and quality for the award of the degree of Bachelor of Electronics Engineering Technology with Honours.

Signature	:	
Supervisor Name	:	DR. AIMAN ZAKWAN BIN JIDIN
Date	:	14 February 2025
Signature	:	
Co-Supervisor Name (if any)	:	
Date	:	

DEDICATION

To my dear family,
To my loyal friends,
To my supervisor and mentors,
And to my dearest self,

ABSTRACT

This project aims to develop and implement a Memory Built-In Self-Test (MBIST) controller in an FPGA device using the Improved March AZ1 algorithm. MBIST is essential for testing memory components within a System-on-Chip (SoC) to ensure they are free of faults. The numbers of test operations increases and results in a higher complexity to detect more faults in the test and will increases the cost and hardware area overhead. The March AZ algorithm was selected for its efficiency in detecting various memory fault models. To overcome more faults in the test, the numbers of test operations increases and results in a higher complexity. The project is proposed to improve the March AZ algorithm's fault coverage as the March AZ1 algorithm while maintaining its complexity is equal to the complexity of 13N. The March AZ1 algorithm is improved by rearranging the test sequences of the March AZ algorithm.Extensive testing on fault-free and fault-injected memory models will be demonstrate the controller's effectiveness in detecting a wide range of memory faults, validating its reliability. The project is successfully implement an effective MBIST solution in FPGA, providing a valuable tool for ensuring memory reliability in SoCs. Future work could be focus on further optimizing the MBIST controller and adapting the algorithm for different memory technologies.

ABSTRAK

Projek ini bertujuan untuk membangunkan dan melaksanakan pengawal Ujian Kendiri Terbina Dalam Memori (MBIST) dalam peranti FPGA menggunakan algoritma AZ1 Mac yang Diperbaiki. MBIST adalah penting untuk menguji komponen memori dalam System-on-Chip (SoC) untuk memastikan ia bebas daripada kerosakan. Bilangan operasi ujian meningkat dan menghasilkan kerumitan yang lebih tinggi untuk mengesan lebih banyak kerosakan dalam ujian dan akan meningkatkan kos dan overhed kawasan perkakasan. Algoritma AZ Mac dipilih untuk kecekapannya dalam mengesan pelbagai model kerosakan memori. Untuk mengatasi lebih banyak kesilapan dalam ujian, bilangan operasi ujian meningkat dan menghasilkan kerumitan yang lebih tinggi. Projek ini dicadangkan untuk menambah baik liputan kesalahan algoritma AZ Mac kerana algoritma AZ1 Mac sambil mengekalkan kerumitannya adalah sama dengan kerumitan 13N. Algoritma AZ1 Mac dipertingkatkan dengan menyusun semula urutan ujian algoritma AZ Mac. Pengujian meluas pada model ingatan bebas kesalahan dan disuntik kesalahan akan menunjukkan keberkesanan pengawal dalam mengesan pelbagai kesalahan memori, mengesahkan kebolehpercayaannya. Projek ini berjaya melaksanakan penyelesaian MBIST yang berkesan dalam FPGA, menyediakan alat yang berharga untuk memastikan kebolehpercayaan memori dalam SoC. Kerja masa depan boleh difokuskan untuk mengoptimumkan lagi pengawal MBIST dan menyesuaikan algoritma untuk teknologi memori yang berbeza.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my supervisor, Dr. Aiman Zakwan Bin Jidin for their precious guidance, words of wisdom and patient throughout this project.

I am also indebted to Universiti Teknikal Malaysia Melaka (UTeM) for the financial support through DR Aiman Zakwan Bin Jidin which enables me to accomplish the project. Not forgetting my fellow colleague, Kok Chi Zhao for the willingness of sharing his thoughts and ideas regarding the project.

My highest appreciation goes to my parents, parents in-law, and family members for their love and prayer during the period of my study. An honourable mention also goes to Shermine Ooi Wei San for all the motivation and understanding.

Finally, I would like to thank all the staffs at the FTKEK, fellow colleagues and classmates, the Faculty members, as well as other individuals who are not listed here for being co-operative and helpful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

MBIST	-	Memory Built-In-Self-Test
SAF	-	Stuck-At Fault
TF	-	Transition Fault
RDF	-	Read Disturb Fault
IRF	-	Inversion Read Fault
DRDF	-	Deceptive Read Disturb Fault
WDF	-	Write Disturb Fault
CFtr	-	Transition Coupling Fault
CFdrd	-	Deceptive Read Destructive Fault
CFwd	-	Write Desturctive Fault

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Research Background

Memory built-in self-test (MBIST) is a method for testing embedded memories on chips that is quite popular since it is a short testing time at a low cost [1]. Because of its ability to self-test and self-check test replies, an expensive external tester is not necessary. Furthermore, it has the ability to run numerous tests concurrently on various memory blocks. Its fault coverage (FC) and test method complexity are the key determinants of its efficiency. The amount of test operations carried out on N memory cells is represented by the test complexity. The longer the test, the more difficult it is, consequently, the chip test's overall duration and production costs rise. MBIST is a widely used technique because it is has a short testing time at a low cost [1].

March test algorithms are a family of memory test algorithms used to detect the faults in memory cells. The name "March" comes from the way these algorithms "march" through the memory addresses, performing write and read operations in a systematic manner. They are designed to detect various fault models such as stuck-at faults, transition faults, coupling faults, and neighborhood pattern sensitive faults [2].

The March AZ2 algorithm and March AZ1 algorithms is the new test algorithms. The March AZ2 algorithm's test sequence was simplified to ⇳ (w0); ⇓ (w0, r0); ⇑ (r0, w1, w1, r1); ⇑ (r1, w0); ⇓ (r0, w1, w1, r1); ⇑ (r1) while the March AZ1 algorithm's test sequences was simplified to ⇳ (w0); ⇓ (w1); ⇑ (w1, r1, r1, w0); ⇑ (w0, r0); ⇑ (r0, w1, w1, r1); ⇑ (r1). For March AZ2 algorithm have a fault coverage (FC) of 83.3%

(detect 30 out of 36 possible fault) with 14N complexity but f or March AZ1 algorithm have a fault coverage (FC) of 80.6% (detect 29 out of 36 possible fault) with 13N complexity [1]. Therefore, this project is proposed with a motivation to improve the March AZ1 to match the March AZ2 fault coverage while maintaining its complexity lower or same to 14N [1].

1.2 Problem Statement

The most use of algorithm are the March algorithms test. By using the March algorithms, the costs and chip area overhead can be reduced with a lower complexity. To detect more faults in the test, the numbers of test operations increases and results in a higher complexity. This will also increases the cost and hardware area overhead [2]. Many 10N and 14N complexity test algorithms are available to produce memory testing within low testing time but most of them have poor coverage of many faults such as Write Disturb Fault (WDF), Data retention fault (DRF) and their coupling faults. So, the March AZ1 algorithms were used to provide an excellent fault coverage while having the complexity at 13N. However, March AZ1 algorithms have a slightly lower Transition Coupling Fault (CFtr) which is 5/8 (62.5%). Therefore, this project focuses on improving the March AZ1 algorithm's fault coverage while maintain the same complexity of 13N [2].

1.3 Project Objective

The proposed project embarks on the following objectives:

I. To develop an improved March AZ1 test algorithm to match the March AZ2 test algorithm fault coverage while maintaining its complexity lower than March AZ2 test algorithm.

II. To implement a Memory BIST controller in FPGA using the improved March AZ1 as the test algorithm.

III. To evaluate the fault coverage and complexity of the improved March AZ1 algorithm through tests and analysis.

1.4 Scope of Project

In order the achieve the project objectives mentioned in Section 1.3, the following works are involved.

I. The March AZ1 was improved by rearranging the test sequences to provide the higher fault coverage as the March AZ1 algorithm which is with the same complexity of 13N.

II. The March AZ1 algorithm and improved March AZ1 algorithm were applied as the UDA in the MBIST controller. The MBIST insertion process was done using Siemens Tessent MemoryBIST software as the tool. The MBIST insertion was targeted for a 1 kB SRAM as the memory model to be tested.

III. The improved March AZ1 algorithm were simulate in the QuestaSim software on both fault-free and fault-injected model. The waveform was captured as the result.

IV. The generated MBIST circuit was implemented in FPGA development board by using Altera Quartus Design software, which synthesize the design before fitting it into the FPGA configuration and generating the bitstream file to be programmed in the FPGA.

V. The tests using the implemented MBIST in FPGA was conducted on a fault-free memory model to validate its functionality

The proposed project functionality evaluation was done by examining the outputs produced from the FPGA experimental tests that will be captured and displayed by the integrated SignalTap logic analyzer.

1.5 Report Outline

This report consists of 5 chapters. Chapter 2 reviews on the semiconductor memories that used in the System-on-Chip (SoC) designs, memory fault models, Memory Built-In Self Test (MBIST), MBIST test algorithms and the existing test algorithms.Chapter 3 reviews on the project design, project planning and project development for the report. Chapter 3 also reviews about the improved March AZ1 algorithm and the software and hardware used in this project. Chapter 4 reviews about the test results for the fault-free memory models and fault-injected memory models of improved March AZ1 algorithm. Chapter 4 also mention about the test on the Altera Quartus using FPGA and the limitation for this project. Chapter 5 states the summary for this project for each objective listed.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter comprehensive reviews about the memories used in SoC, the on-chip memory testing using MBIST, and the existing test algorithms. Section 2.2 review on the semiconductor memories while Section 2.3 describe about the memory fault models. Section 2.4 describes about the Memory Built-In Self Test (MBIST) while section 2.4. Next, section 2.5 review about the MBIST test algorithm while section 2.6 reviews about the existing test algorithms.

2.2 Semiconductor Memories

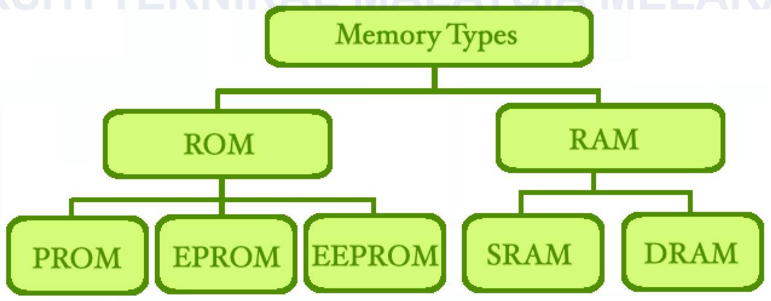


Figure 2.1 Types of memories

Embedded memories in System-on-Chip (SoC) designs refer to memory components that are integrated directly onto the same silicon die as the other components of the system. These memories serve as crucial storage elements for data and instructions within the SoC, enabling various functions and applications to operate efficiently [3].

Random Access Memory (RAM) is a volatile memory type, meaning its contents are lost when the power is turned off. It is used to temporarily store data and program instructions that the Central Processing Unit (CPU) needs to access quickly during the execution of programs [3]. RAM is characterized by its fast read and write speeds, making it ideal for tasks that require frequent data access and manipulation. ROM is a non-volatile memory type, meaning its contents are retained even when the power is turned off. It is used to store permanent or semi- permanent data and program instructions that are not expected to change frequently during the operation of the system [3].

There were some types of Embedded Memories which is Static Random-Access Memory (SRAM), Dynamic Random-Access Memory (DRAM) and Non-Volatile Memories. SRAM is often used for cache memory within SoCs due to its fast access times and low power consumption. It provides high-speed access to frequently accessed data, enhancing the overall performance of the system. DRAM is used for main memory in SoCs, providing larger storage capacity compared to SRAM but with slightly slower access times. It serves as the primary storage for program instructions and data during the execution of applications. SoCs may also include non-volatile memory components like Flash memory or Electrically Erasable Programmable Read-Only Memory (EEPROM) for storing firmware, boot code, or configuration data that needs to persist across power cycles [4].

The embedded memories are typically integrated directly onto the SoC die using specialized manufacturing processes. This integration reduces the need for external memory chips, saving board space and reducing power consumption. Interconnect structures within the SoC connect the embedded memories to the CPU, peripherals, and other components, enabling data transfer and access [5].

Given the critical role of embedded memories in SoCs, thorough testing and verification processes are essential to ensure their reliability and functionality [5]. Techniques like Memory Built-In Self-Test (MBIST) are commonly employed to enable memories to test themselves for faults and defects autonomously. Advanced verification methodologies, including simulation, emulation, and formal verification, are used to validate memory designs and ensure they meet performance, power, and reliability requirements [6].

In summary, embedded memories are essential components of SoCs, providing storage for data and instructions and influencing the overall performance, power consumption, and security of the system. Designing, integrating, and testing embedded memories require careful consideration of various factors to ensure the reliability, functionality, and security of SoCs in diverse applications [6].

2.3 Memory fault models

Faults in digital systems are broadly categorized into static and dynamic faults, each representing different types of malfunctions that can occur in a system [7]. Static faults are those where the failure mode is constant and does not change over time [7]. Examples include stuck-at faults, where a signal line is permanently stuck at a logical high (1) or low (0) value, and open faults, where a circuit line is disconnected and thus always reads as an undefined state. These faults are relatively straightforward to detect with static test patterns that do not rely on changing conditions. Static faults typically result from physical defects in the hardware, such as manufacturing imperfections, and are often identified through direct inspection or simple test algorithms [8].

In contrast, dynamic faults occur under specific conditions or sequences of operations, making them more challenging to detect [5]. Dynamic faults are often influenced by factors like operational timing, environmental conditions, and the specific usage pattern of the device. Detecting these faults requires more sophisticated testing methodologies that involve sequences of operations, varying conditions, and timing analysis. Dynamic faults are critical to identify because they can lead to intermittent and unpredictable system failures, which are harder to diagnose, and fix compared to static fault [5].

This project focuses on the detection of static faults, which are faults that remain constant and do not change over time. These include common issues like stuck-at faults, where a signal line is permanently fixed at a logical high or low value, and open faults, where a circuit line is disconnected and consistently reads as an undefined state. By employing specific test algorithms designed to identify these static faults, the project aims to ensure the reliability and correctness of hardware systems, addressing critical defects that can result from manufacturing imperfections and other static conditions.

There are many type of static fault which are [5]:

1. Stuck-at Fault (SAF): A stuck-at fault occurs when a signal line or node in a digital circuit is consistently stuck at either a logic high ('1') or a logic low ('0') state. This fault remains persistent regardless of the input or operation. SAFs are typically caused by defects in the manufacturing process, such as material impurities or physical damage, and are relatively straightforward to detect using test patterns that attempt to force the line to both states and then verify its response.

2. Transition Fault (TF): Transition faults manifest as failures in the transition of a digital signal from one logic state to another within the expected timeframe. There are two types: rising transition faults (0 to 1) and falling transition faults (1 to 0). These faults can occur due to various reasons, including timing violations, signal integrity issues, or manufacturing irregularities. Detecting TFs involves applying test patterns that aim to induce transitions and verifying if they occur as expected.

3. Read Disturb Fault (RDF): RDFs arise in non-volatile memory devices when the act of reading data repeatedly from a memory cell disturbs the contents of nearby cells. This disturbance can result in unintended changes in adjacent memory locations. RDFs are particularly relevant in flash memory technologies and can degrade device reliability over time due to wear-out mechanisms.

4. Inversion Read Fault (IRF): IRF occurs when reading a memory cell results in the incorrect inversion of the stored data. For instance, reading a '1' might return a '0' and vice versa. This fault can be caused by various factors,

including transistor leakage or cross-coupling effects, and can lead to data integrity issues in memory systems.

5. Deceptive Read Disturb Fault (DRDF): DRDFs occur when reading a memory cell disturbs the data in neighboring cells temporarily, leading to incorrect readouts during specific read operations. Unlike RDFs, which cause permanent changes, DRDFs only affect the read operation temporarily and may not be immediately detectable after the read operation.

6. Write Disturb Fault (WDF): WDFs arise when writing data to a memory cell inadvertently alters the contents of nearby cells due to electrical interference or coupling effects. These faults can occur in various memory technologies, including DRAM and flash memory, and can lead to data corruption or integrity issues if left undetected.

7. Transition Coupling Fault (CFtr): Occurs when a transition in one memory cell triggers an unintended transition in a neighboring cell.

8. Deceptive Read Destructive Fault (CFdrd): Involves a read operation that disturbs the data in a neighboring cell, leading to incorrect readouts and potentially permanent data corruption.

9. Write Destructive Fault (CFwd): Arises when writing data to one memory cell inadvertently alters the contents of another cell. CFs require comprehensive testing strategies to detect interactions between memory cells and mitigate their impact on system reliabily.

2.4 Memory Built-In Self Test (MBIST)

MBIST is a design technique used in integrated circuits (ICs) to ensure the reliability and functionality of embedded memories, such as SRAM (Static Random-Access Memory) or DRAM (Dynamic Random-Access Memory). In MBIST, the memory is equipped with circuitry that allows it to test itself for faults and defects without requiring external test equipment. This self-testing capability is crucial for detecting and diagnosing faults that may occur during the operation of the device, such as manufacturing defects, aging-related issues, or environmental factors [8].

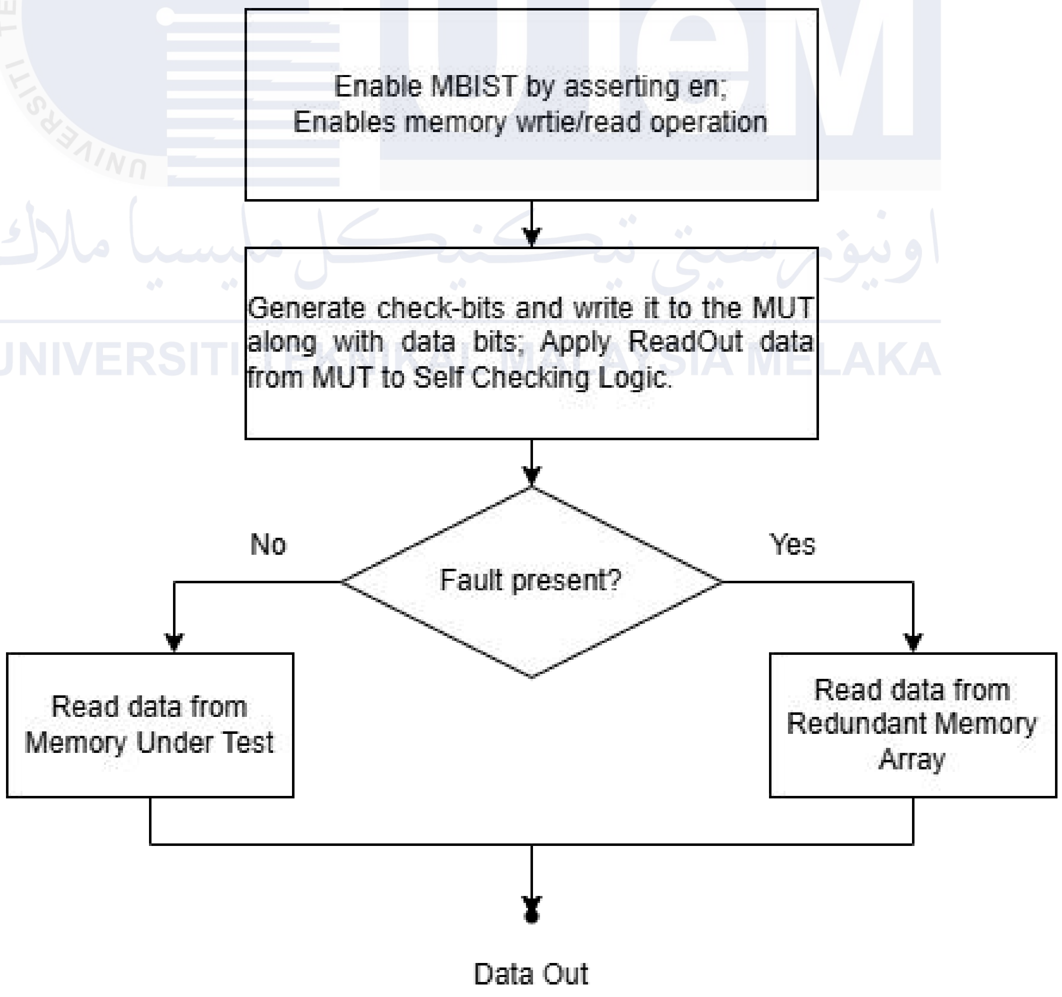

Enable MBIST by asserting en;
Enables memory wrtie/read operation
Generate check-bits and write it to the MUT along with data bits; Apply ReadOut data from MUT to Self Checking Logic.
Fault present?
No
Yes
Read data from Memory Under Test
Read data from Redundant Memory Array
Data Out

Figure 2.2 Flowchart of MBIST

MBIST typically involves generating test patterns and applying them to the memory array, then comparing the expected results with the actual results to identify any discrepancies [9]. It helps in identifying faults such as stuck-at faults(where a bit is always at a high or low logic level), transition faults (where a bit fails to switch its value), and coupling faults (where adjacent bits interfere with each other)[9]. Overall, MBIST plays a critical role in ensuring the reliability and quality of memory components in modern integrated circuits.

The principle of working of MBIST involves several key steps [10]:

1. Test Pattern Generation: The MBIST controller generates a set of test patterns designed to thoroughly exercise the memory array. These patterns are carefully crafted to detect various types of faults, including stuck-at faults, transition faults, and coupling faults.

2. Test Pattern Application: The generated test patterns are applied to the memory array by configuring the memory's address and data input/output paths accordingly. The memory controller coordinates this process to ensure that each test pattern is applied correctly.

3. Data Comparison: As the test patterns are applied, the memory controller compares the expected output data (based on the test patterns) with the actual output data obtained from the memory array. Any discrepancies between the expected and actual data indicate potential faults or defects within the memory.

4. Fault Identification: When discrepancies are detected, the MBIST controller identifies the specific locations within the memory array where faults may be present. This information is crucial for diagnosing the root causes of the faults.

5. Reporting: The results of the MBIST self-test are typically logged and reported to higher- level system components. This information allows system-level firmware or

software to make informed decisions about the operational status and reliability of the memory.

Siemens Tessent MemoryBIST software streamlines MBIST generation by automating the creation of tailored MBIST structures for specific memory architectures and test requirements [11]. Designers input parameters such as memory size, organization, and desired fault coverage levels, and the software utilizes sophisticated algorithms to generate the necessary MBIST logic, including test pattern generators, response analyzers, and control circuitry. This automated process ensures efficient integration of MBIST into the semiconductor design, enhancing memory testing capabilities while minimizing development time and effort, ultimately improving product quality and reliability [12].

There were some type of algorithm such as non-linear algorithm, linear algorithm and classical test algorithm. For non-linear algorithm, a non-linear algorithm is a computational process where the relationship between input size and computation time or resource usage is not proportional or directly scalable, often involving complexities higher than linear, such as quadratic, polynomial, or exponential [13]. Unlike linear algorithms, which exhibit a straight-line growth pattern as inputs increase, non-linear algorithms may involve nested loops, recursive calls, or combinatorial processes, leading to growth rates that escalate rapidly with larger inputs. These algorithms are essential for solving complex problems in fields like optimization, graph theory, and artificial intelligence, where interactions among elements do not follow a simple, direct pattern and require more sophisticated approaches to manage [13].

Next, a linear algorithm is a computational process in which the time or resources required to complete the task increase directly in proportion to the size of the input [14]. This means that if the input size doubles, the computation time or resource usage also doubles, resulting in a predictable and manageable growth pattern [14]. Linear algorithms are characterized by their simplicity and efficiency, making them ideal for tasks such as searching, traversing a list, or performing basic arithmetic operations where each step involves a constant amount of work relative to the input size. This predictability and scalability make linear algorithms fundamental in computer science for handling straightforward problem efficiently[14].

2.5 MBIST test algorithms

A classical test algorithm refers to a traditional, well-established method used for assessing the functionality, performance, or reliability of a system or component, often in hardware and software testing [15]. These algorithms follow a predefined sequence of operations designed to systematically evaluate various aspects of the subject under test, such as correctness, efficiency, and fault tolerance [15]. Examples include the binary search for testing sorted arrays, the bubble sort for sorting algorithms, and the basic memory test algorithms like MATS (Modified Algorithmic Test Sequence) and the March test for memory fault detection. Classical test algorithms are foundational in ensuring the integrity and performance of systems, leveraging their simplicity and robustness to provide reliable testing results [15].

The March test algorithms are a family of memory testing algorithms commonly used to test the functionality and reliability of memory components, particularly SRAM (Static Random-Access Memory) [16]. These algorithms systematically walk through memory cells in a specific order, performing read, write, and compare operations. They are designed to detect various types of faults, including stuck-at faults, transition faults, and coupling faults. The name "March" is derived from the initials of its creator, IBM engineer R.H. "Bob" March [17].

The March algorithm is a systematic method used for testing and detecting faults in

Random Access Memory (RAM) [18]. It operates by executing a series of read and write operations on each memory cell in a specific sequence, often traversing the memory in both forward and backward directions [18]. This bidirectional approach helps identify various types of faults such as stuck-at faults, transition faults, coupling faults, and address decoder faults [18]. By meticulously marching through the memory, the algorithm ensures comprehensive fault coverage while maintaining efficiency in the number of operations performed [18].

Designed to be efficient and thorough, March algorithms balance the trade-off between testing time and fault detection capability. An example is the March C- algorithm, which uses a sequence of write, read, and compare operations in both ascending and descending address orders to uncover faults. This methodical process ensures that each cell is tested multiple times under different conditions, enhancing the likelihood of detecting any present faults. Overall, the March algorithm's structured approach and bidirectional testing make it a robust tool for ensuring memory reliability [16].

In a checkerboard pattern, the 1s and 0s are written into different memory locations within the cell array. In order to place each nearby cell in a separate group, the algorithm splits the cells into two alternating groups. The checkerboard pattern is mostly utilized to activate failures brought on by SAF, leakage, and cell-to-cell shorts [19].

2.6 Review on existing test algorithms

The table outlines various test algorithms used for detecting faults in integrated circuits, focusing on their complexity and fault coverage. Basic algorithms like March C- and March CL provide foundational fault detection with complexities of 10N and 12N, respectively, offering comprehensive coverage for stuck-at faults (SAF) and transition faults (TF)[12]. These algorithms use a series of read and write operations to identify faults, with March CL extending its sequences to include additional read operations for improved fault detection [20].

More advanced algorithms, such as PMOVI and March RAW1, increase the complexity to 13N, incorporating sequences that alternate between reading and writing operations [21]. These algorithms extend fault detection to include data retention faults (DRF) and coupling faults (CF), providing more robust fault coverage. March LR and March SR further enhance this approach with a complexity of 14N, focusing on detecting linear read and write errors [22]. The Modified March SR algorithm improves this by increasing coverage for write disturb faults (WDF) and coupling faults through modified read/write sequences [23].

The most comprehensive algorithms, including March AZ1, March AZ2, March- sift, March-ee, March MSS, March LV, March CS, March SS, and March RAW, feature higher complexities ranging from 14N to 26N [24]. These algorithms offer extensive sequences designed to detect a broad spectrum of faults, including subtle and complex ones like coupling faults and write disturb faults [24]. Algorithms like March MSS and March SS, with complexities of 18N and 22N, respectively, provide exhaustive read/write operations to ensure thorough fault detection across all major fault types [24]. March RAW, with the highest complexity of 26N, incorporates the most detailed sequences to achieve complete fault coverage, ensuring the highest reliability by detecting even the most challenging faults [24].

In the domain of memory testing, achieving an optimal balance between test complexity and fault coverage (FC) is essential. Simpler algorithms like March C and March C- offer quick and efficient tests with low complexity (10N-11N), but they fall short in detecting more complex faults, providing only basic fault coverage [25]. Algorithms like March CL and March SR improve on this by including double read operations, enabling partial detection of certain faults like DRDF and CFdrds with moderate complexity (12N-14N) [22]. However, they still miss other fault types such as Write Disturb Faults (WDF) and Write Coupling Faults (CFwds). More complex algorithms such as March-sift and March-ee, despite having higher complexities (17N-18N), show limitations due to redundancy in their sequences and lack of specific operations required to detect all fault types comprehensively [26].

March AZ1 and March AZ2 are preferred because they strike a balance between fault coverage and test complexity, making them efficient and effective for practical memory testing [24]. Both algorithms offer high fault coverage comparable to more complex tests but maintain moderate complexity, avoiding excessive resource and time demands [24]. March AZ1 uses a sequence focused on efficient fault detection through repeated write and read operations, while March AZ2 adjusts the order slightly to achieve similar results. Both algorithms are designed to handle a wide range of faults effectively, making them suitable for general-purpose memory testing where comprehensive fault detection is necessary without incurring prohibitive complexity [24].

Table 2.1 shows the examples of test algorithms with their complexity and test operation sequences while Table 2.2 shows the examples of test algorithms with the complexity and fault coverage. The RDF and IRF FC are represented by SAF coverage since they have similar detection requirement [1].

Table 2.1 Example of several March test algorithm with their sequences

Test Algorithm	**Test Complexity**	**Test Operation Sequences**
March C- [26]	10*N*	⇕ (w0); ⇑ (r0, w1);⇑ (r1, w0); ⇓ (r0,w1); ⇓ (r1, w0);⇕ (r0)
March CL [27]	12*N*	⇕ (w0); ⇑ (r0, w1);⇕ (r1); ⇑ (r1, w0);⇓ (r0, w1); ⇕ (r1);⇓ (r1, w0); ⇕ (r0)
PMOVI [28]	13*N*	⇕ (w0); ⇑ (r0, w1,r1); ⇑ (r1, w0, r0);⇓ (r0, w1, r1); ⇓ (r1,w0, r0);
March RAW1 [29]	13*N*	⇕ (w0); ⇕ (w0, r0);⇕ (r0); ⇕ (w1, r1);⇕ (r1); ⇕ (w1, r1);⇕ (r1); ⇕ (w0, r0);⇕ (r0)
March LR [30]	14*N*	⇕ (w0); ⇓ (r0, w1); ⇑ (r1, w0, r0, w1);⇑ (r1, w0); ⇑ (r0, w1, r1, w0); ⇑ (r0)
March SR [31]	14*N*	⇕ (w0); ⇑ (r0, w1, r1, w0); ⇑ (r0, r0); ⇑ (w1); ⇓ (r1, w0, r0, w1); ⇓ (r1, r1)
Modified March SR [32]	14*N*	⇕ (w0); ⇑ (r0, w0, r0, w1); ⇑ (r1, r1); ⇑ (w1); ⇓ (r1, w0, r0, w0); ⇓ (r0, r0)
March C+ [33]	14*N*	⇕ (w0); ⇑ (r0, w1, r1); ⇑ (r1, w0, r0); ⇓ (r0, w1, r1); ⇓ (r1, w0, r0); ⇕ (r0)
March AZ1 [34]	13*N*	⇕ (w0); ⇓ (w1); ⇑ (w1, r1, r1, w0); ⇑ (w0, r0); ⇑ (r0, w1, w1, r1); ⇑ (r1);

Test Algorithm	**Test Complexity**	**Test Operation Sequences**
March AZ2 [34]	14*N*	⇕ (w0); ⇓ (w0, r0); ⇑ (r0, w1, w1, r1); ⇑ (r1, w0); ⇓ (r0, w1, w1, r1); ⇑ (r1);
March-sift [35]	17*N*	⇕ (w0); ⇑ (r0, w1); ⇓ (r1, w0, r0); ⇑ (r0, w1); ⇑ (r1, w0); ⇓ (r0, w0, r0); ⇑ (r0, w1, r1); ⇕ (r0)
March-ee [36]	18*N*	⇑ (w0); ⇑ (r0, w1, r1); ⇑ (r1, w0, r0); ⇑ (r0, w1); ⇓ (r1, w0, r0); ⇑ (r0, w0); ⇓ (r0, w1, r1); ⇑ (r1)
March MSS [37	18*N*	⇕ (w0); ⇑ (r0, r0, w1, w1); ⇑ (r1, r1, w0, w0); ⇓ (r0, r0, w1, w1); ⇓ (r1, r1, w0, w0); ⇕ (r0)
March LV [38]	18*N*	⇕ (w0); ⇑ (r0, w1, w1, r1); ⇑ (r1, w0, w0, r0); ⇓ (r0, r0, w1, r1); ⇓ (r1, r1, w0, r0); ⇓ (r0)
March CS [39]	20*N*	⇕ (w0); ⇑ (w0, r0, w1, r1); ⇕ (w1); ⇑ (w1, r1, w0, r0); ⇓ (w0, r0, w1, r1); ⇓ (w1, r1, w0, r0); ⇕ (w0, r0)
March SS [40]	22*N*	⇕ (w0); ⇑ (r0, r0, w0, r0, w1); ⇑ (r1, r1, w1, r1, w0); ⇓ (r0, r0, w0, r0, w1); ⇓ (r1, r1, w1, r1, w0); ⇕ (w0);
March RAW [29]	26*N*	⇕ (w0); ⇑ (r0, w0, r0, r0, w1, r1); ⇑ (r1, w1, r1, r1, w0, r0); ⇓ (r0, w0, r0, r0, w1, r1); ⇓ (r1, w1, r1, r1, w0, r0); ⇕ (r0)

Table 2.2 Example of several March test algorithms with their fault coverage

Test Algorithm	**Test Complexity**	**SAF**	**TF**	**DRDF**	**WDF**	**CFtr**	**CFdrd**	**CFwd**
March C- [26]	10*N*	100%	100%	-	-	100%	-	-
March CL [27]	12*N*	100%	100%	50%	-	100%	25%	-
PMOVI [28]	13*N*	100%	100%	100%	-	100%	75%	-
March RAW1 [29]	13*N*	100%	100%	100%	100%	50%	75%	50%
March LR [30]	14*N*	100%	100%	-	-	100%	-	-
March SR [31]	14*N*	100%	100%	100%	-	100%	50%	-
Modified March SR [32]	14*N*	100%	100%	100%	100%	50%	62.5%	50%
March C+ [33]	14*N*	100%	100%	100%	-	100%	100%	-
March AZ1 [34]	13*N*	100%	100%	100%	100%	62.5%	75%	75%
March AZ2 [34]	14*N*	100%	100%	100%	100%	75%	75%	75%
March-sift [35]	17*N*	100%	100%	100%	50%	75%	62.5%	25%

Test Algorithm	**Test Complexity**	**SAF**	**TF**	**DRDF**	**WDF**	**CFtr**	**CFdrd**	**CFwd**
March-ee [36]	18N	100%	100%	100%	50%	100%	100%	25%
March MSS [37	18N	100%	100%	100%	100%	100%	100%	100%
March LV [38]	18N	100%	100%	100%	100%	100%	100%	50%
March CS [39]	20N	100%	100%	100%	100%	100%	100%	100%
March SS [40]	22N	100%	100%	100%	100%	100%	100%	100%
March RAW [29]	26N	100%	100%	100%	100%	100%	100%	100%

In the realm of memory testing, achieving a balance between test complexity and fault coverage (FC) is crucial. Simple algorithms like March X and March Y offer quick and efficient tests but fall short in detecting a wide range of faults, limiting their fault coverage[1]. On the other hand, more complex algorithms such as March LV and March EE provide very high fault coverage but at the cost of significantly increased test complexity and duration. This high complexity can be prohibitive in terms of time, computational resources, and practicality for large-scale or real-time applications [1].

March AZ1 and March AZ2 stand out as preferred choices because they strike an optimal balance between test complexity and fault coverage [1]. Both algorithms are designed to efficiently detect a wide array of faults, offering high fault coverage comparable to more complex algorithms, but without the excessive overhead [1]. This balance makes them particularly suitable for practical memory testing scenarios where both comprehensive fault detection and efficiency are necessary [1]. By maintaining moderate complexity, these algorithms ensure that the testing process remains feasible and cost-effective [1].

Comparing March AZ1 and March AZ2, both algorithms provide similar levels of fault coverage and share the common strength of balancing efficiency with thoroughness [1]. March AZ1 uses a sequence of write and read operations (⇕ (w0);⇓ (w1); ⇑ (w1, r1, r1, w0); ⇑ (w0, r0); ⇑ (r0, w1, w1, r1); ⇑ (r1);), while March AZ2 uses a slightly different sequence (⇕ (w0); ⇓ (w0, r0); ⇑ (r0, w1, w1, r1); ⇑ (r1, w0);⇓ (r0, w1, w1, r1); ⇑ (r1);) [1].The primary difference lies in the specific order of operations, which may impact their implementation details and performance in specific contexts [1]. However, both are well-suited for general-purpose memory testing, offering reliable fault detection without imposing excessive test complexity. The choice between the two can be based on specific application requirements or preferences in testing methodology [1].

2.7 Summary

In Chapter 2, the primary focus is on the performance evaluation of different test algorithms used in our system. The analysis is centered around two main algorithms: March AZ1 and AZ2 algorithms. The chapter divided into various aspects of these algorithms, including their complexity, fault coverage (FC), and overall fault coverage. For March AZ1 algorithm, the fault coverage of March AZ1 algorithms is lower compared to March AZ2 algorithms, which limits its effectiveness in detecting all potential faults and the complexity of the AZ1 algorithm is relatively low, making it computationally efficient and suitable for systems with limited resources. Its complexity is denoted as $\leq 14N$.

Basically, this project identify the weakness in fault detection by the March AZ algorithms. Then, we rearrange the chosen algorithms test operations to increase the coverage of that particular fault type without reducing its overall fault coverage.

CHAPTER 3

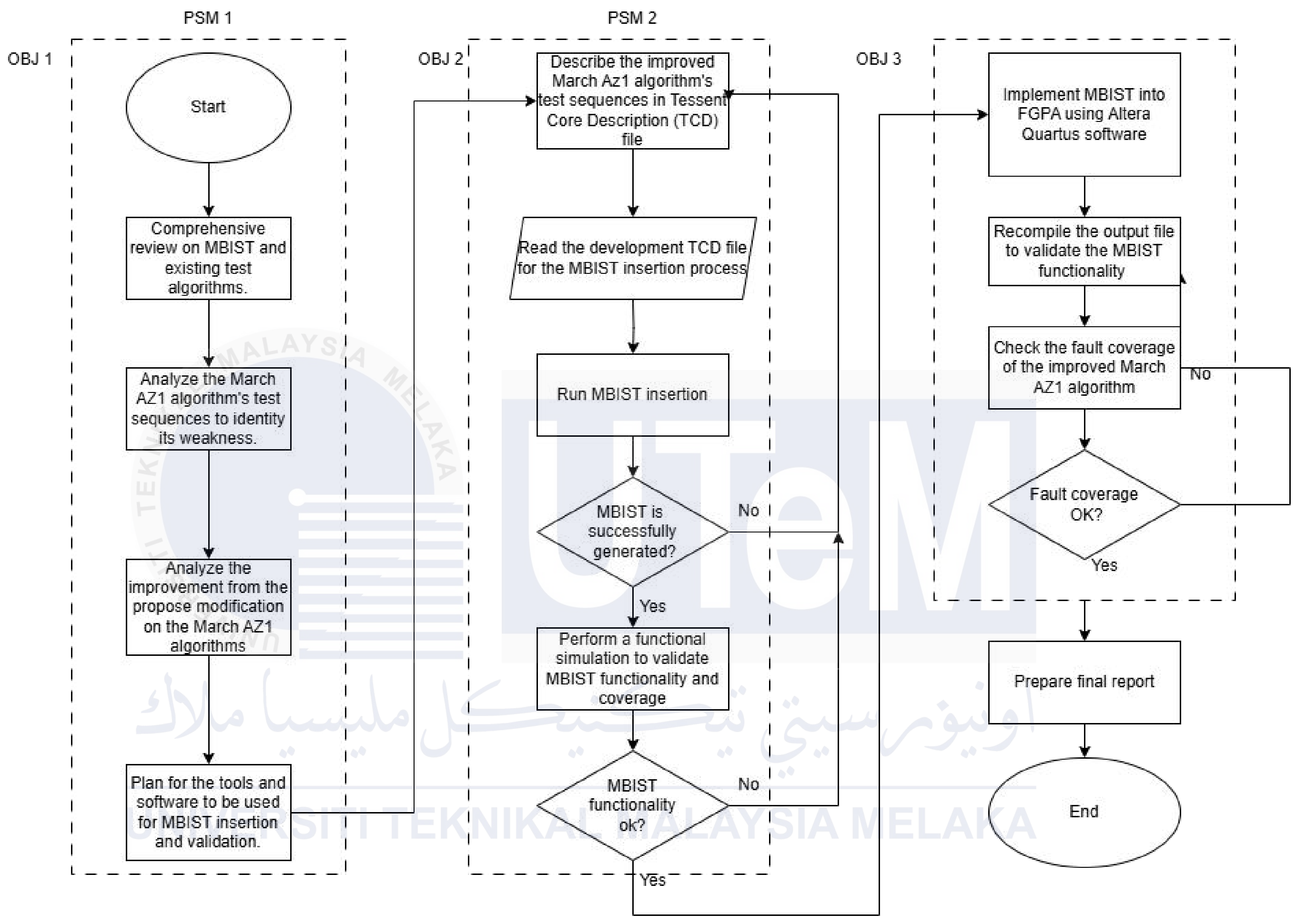
METHODOLOGY

3.1 Introduction

This chapter comprehensively describes the method for the implementation of Memory BIST (MBIST) Controller in FPGA using the Improved March AZ1 as the Test Algorithm. Section 3.2 have 2 part which section 3.2.1 describe about the project execution flow and section 3.2.2 describe about the project planning. Therefore, section 3.3 mentions about the Improved March AZ1 Algorithm while section 3.4 describe about the project development. For section 3.5 is the lists of the software and hardware used in this project.

3.2 Project Design

3.2.1 Project Execution Flow



PSM 1
PSM 2
OBJ 1
OBJ 2
OBJ 3
Start
Comprehensive review on MBIST and existing test algorithms.
Analyze the March AZ1 algorithm's test sequences to identity its weakness.
Analyze the improvement from the propose modification on the March AZ1 algorithms
Plan for the tools and software to be used for MBIST insertion and validation.
Describe the improved March Az1 algorithm's test sequences in Tessent Core Description (TCD) file
Read the development TCD file for the MBIST insertion process
Run MBIST insertion
MBIST is successfully generated?
No
Yes
Perform a functional simulation to validate MBIST functionality and coverage
MBIST functionality ok?
No
Yes
Implement MBIST into FGPA using Altera Quartus software
Recompile the output file to validate the MBIST functionality
Check the fault coverage of the improved March AZ1 algorithm
Fault coverage OK?
No
Yes
Prepare final report
End

Figure 3.1 Project Execution Flowchart

The flowchart in Figure 3.1 illustrates the process of developing and validating an improved memory built-in self-test (MBIST) algorithm, specifically focusing on enhancing the March AZ1 algorithm. The process begins with a comprehensive review of MBIST methodologies and existing test algorithms. Following this review, the weaknesses of the March AZ1 algorithm's test sequences are analyzed. Based on this analysis, modifications are proposed, and the potential improvements from these modifications are evaluated. Finally, the necessary tools and software for MBIST insertion and validation are planned.

The second phase, denoted as PSM 2, involves the practical implementation of the improved March AZ1 algorithm. The process starts by describing the improved test sequences in the Tessent Core Description (TCD) file. The development TCD file was then read for the MBIST insertion process. If MBIST insertion is successful, a functional simulation is performed to validate the generated MBIST functionality. If the MBIST functionality is satisfactory, the process moves forward; otherwise, adjustments are made, and the process is repeated.

The final phase involves implementing the validated MBIST into an FPGA using Quartus software. Once implemented, the fault coverage of the improved March AZ1 algorithm was checked. If the fault coverage meets the required standards, a final report is prepared, concluding the process. If the fault coverage is inadequate, further improvements and validations are conducted until satisfactory coverage is achieved. This iterative process ensures that the MBIST algorithm is robust and effective in identifying faults.

The implementation of a MBIST controller in an FPGA using the Improved March AZ1 algorithm begins with thorough planning and requirements analysis. This phase involves defining project objectives, allocating resources, and setting a detailed timeline. The requirements analysis focuses on understanding the functional and non-functional needs of the BIST controller, including the specific memory characteristics and performance criteria. Following this, the improved March AZ1 algorithm was studied and optimized for hardware implementation to ensure efficient fault coverage and minimal resource utilization.

3.2.2 Project Planing

The proposed project is planned for a total duration of 28 weeks that comprise both PSM 1 and PSM 2. The Gantt chart in Appendix 1 provides the project activities and timeline, while Appendix 2 shows the project milestones.

In Appendix 1 and Appendix 2, it shows all the activities done in PSM 1 and activities that done in PSM 2.

In PSM1, form week 1 to week 5 was the comprehensive investigation on March algorithms. Analyze the original and improved March AZ1 algorithm test sequence and fault coverage was done between week 5 to week 7. I plan to search for the tools and FPGA device for MBIST implementation and validation process at week 7 to week 12 and prepare my PSM report in week 10 to week 14.

In PSM2, by synthesizing the MBIST circuitry design and fit it into targeted FPGA device using Altera Quartus software in the week 15-20. In the meanwhile, for week 17 to week 23 were plan to validate the implemented MBIST through FPGA experimental test. Next is to analyze the implement MBIST performance in terms of its fault coverage, speed, and area. The duration for this activity is 5 week which is week 21 to week 25. Finally, the last 2 activity is prepare my final report and presentation. For final report submission was in week 28 and the presentation was located at the final 2nd weeks, which is week 27.

3.3 The Improved March AZ1 Algorithm

With the same 13N complexity and the following test sequence, the recent March AZ1 algorithm is now known as the March AZ1 algorithm with the test sequences of ⇕ (w0); ⇓ (r0, w1); ⇑ (w1, r1, r1, w0, w0); ⇑ (r0); ⇑ (r0, w1, w1, r1); ⇑ (r1).

Table 3.1 The Recent March AZ1 algorithm's fault detection analysis

Fault	**Fault Primitive**	**Detection Status**	**Fault Coverage**
SAF	< 1 / 0 / - >	Yes	2/2 (100%)
	< 0 / 1 / - >	Yes	
TF	< 0w1 / 0 / - >	Yes	2/2 (100%)
	< 1w0 / 1 / - >	Yes	
RDF	< r0 / 1 / 1 >	Yes	2/2 (100%)
	< r1 / 0 / 0 >	Yes	
IRF	< r0 / 0 / 1 >	Yes	2/2 (100%)
	< r1 / 1 / 0 >	Yes	
DRDF	< r0 / 1 / 0 >	Yes	2/2 (100%)
	< r1 / 0 / 1 >	Yes	
WDF	< 0w0 / 1 / - >	Yes	2/2 (100%)
	< 1w1 / 0 / - >	Yes	
CFtr	< 0; 0w1 / 0 / - > *a>v*	No	6/8 (75%)
	< 0; 0w1 / 0 / - > *a<v*	Yes	
	< 1; 0w1 / 0 / - > *a>v*	Yes	
	< 1; 0w1 / 0 / - > *a<v*	Yes	
	< 0; 1w0 / 1 / - > *a>v*	Yes	
	< 0; 1w0 / 1 / - > *a<v*	No	
	< 1; 1w0 / 1 / - > *a>v*	Yes	
	< 1; 1w0 / 1 / - > *a<v*	Yes	
CFdrd	< 0; r0 / 1 / 0 > *a>v*	Yes	6/8 (75%)
	< 0; r0 / 1 / 0 > *a<v*	Yes	
	< 1; r0 / 1 / 0 > *a>v*	No	
	< 1; r0 / 1 / 0 > *a<v*	No	
	< 0; r1 / 0 / 1> *a>v*	Yes	
	< 0; r1 / 0 / 1> *a<v*	Yes	
	< 1; r1 / 0 / 1> *a>v*	Yes	
	< 1; r1 / 0 / 1> *a<v*	Yes	
CFwd	< 0 ; 0w0 / 1 /- > *a>v*	Yes	4/8(50%)
	< 0 ; 0w0 / 1 /- > *a<v*	Yes	
	< 1 ; 0w0 / 1 /- > *a>v*	No	
	< 1 ; 0w0 / 1 /- > *a<v*	No	
	< 0 ; 1w1 / 0 /- > *a>v*	Yes	
	< 0 ; 1w1 / 0 /- > *a<v*	Yes	
	< 1 ; 1w1 / 0 /- > *a>v*	No	
	< 1 ; 1w1 / 0 /- > *a<v*	No	

The recent March AZ1 algorithms have a slightly lower coverage on the CFwd. So, the improved March AZ1 algorithms was by rearranging the sequences of March AZ1 algorithms. The test sequence of the improved March AZ1 algorithm was divided into six test elements, denoted as TE_0 through TE_5, and separated from one another by a semicolon as shown in Table 3.1. Throughout the test, the test components was be carried out in a sequential manner. Before going on to the next TE, all test procedures specified in TE must be completed on all memory cells. Its 13N complexity is also explained by the fact that all N memory cells require 13 read or write operations to be completed.

The TCD file for improved March AZ1 algorithm is list at the Appendix 3 while the TCD file to run the tessent shell software is alsolist in the Appendix 4.

Table 3.2 The Improved March AZ1 algorithm descriptions

Test Element	**Test Sequence**	**Test Description**
TE_0	⇳ (w0)	All cells are written to 0.
TE_1	⇓ (w1)	All cells are written to 1 in descending address order.
TE_2	⇑ (w1, r1, r1 w0)	All cells are sequentially written to 1, read twice (expecting a 1 at the output), and written to 0 in ascending address order.
TE_3	⇑ (w0, r0)	All cells are sequentially written to 0 before being read (expecting a 0 at the output) in ascending address order.
TE_4	⇑ (r0, w1, w1 r1)	All cells are sequentially read (expecting 0), written to 1 twice, and reread (expecting 1) in ascending address order.
TE_5	⇑ (r1)	All cells are read (expecting 1) in ascending address order.

Using a specially designed fault detection analyzer that finds the sensitizer and detector pairings for each targeted FP in the test sequence [1], a fault detection analysis was performed on the March AZ1 algorithm. Based on their detection requirements as outlined in Chapter 2, it identified all potential pairs of sensitizer and detector for each detectable FP detected inside the analyzed test sequence, from the first test operation defined in TE_0 to the last test operation in TE_5. The sensitizer and detector pairs for each FP found throughout the analysis's test sequence of the March AZ1 algorithm are displayed in

Table 3.2. The sensitizing or detecting operation for each FP was found at the jth test operation in TEi, as indicated by the TEi-j notations. Multiple pairings of sensitizer-detector were found in some FPs during the test process. Table 3.2 illustrates that:

- All targeted SCFs are detectable since their FPs have at least one identified sensitizer-detector pair. So, the improved March AZ1 algorithm offers 100% of all SCFs.
- The fault analyzer identified the sensitizer-detector pairs for 5 FPs of CFtr. Hence, CFtr coverage equals 62.5% (5 detectable FPs out of 8).
- The fault analyzer identified the sensitizer-detector pairs for 6 FPs of each CFdrd and CFwd. Hence, the CFdrd and CFwd coverages equal 75% (6 detectable FPs out of 8).
- The fault detection analysis derived the expected fault coverage by the improved March AZ1 algorithm.

Table 3.3 The Improved March AZ1 algorithm's fault detection analysis

Fault	**Fault Primitive**	**Identified (Sensitizer,Detector)**	**Detection Status**	**Fault Coverage**
SAF	< 1 / 0 / - >	(TE_{2-1}, TE_{2-2}),(TE_{4-3},TE_{4-4})	Yes	2/2 (100%)
	< 0 / 1 / - >	(TE_{2-5}, TE_{3-1})	Yes	
TF	< 0w1 / 0 / - >	(TE_{1-1}, TE_{2-2}),(TE_{4-2},TE_{4-4})	Yes	2/2 (100%)
	< 1w0 / 1 / - >	(TE_{2-4}, TE_{3-1})	Yes	
RDF	< r0 / 1 / 1 >	(TE_{2-1}, TE_{2-2}),(TE_{4-3},TE_{4-4})	Yes	2/2 (100%)
	< r1 / 0 / 0 >	(TE_{2-5}, TE_{3-1})	Yes	
IRF	< r0 / 0 / 1 >	(TE_{2-1}, TE_{2-2}),(TE_{4-3},TE_{4-4})	Yes	2/2 (100%)
	< r1 / 1 / 0 >	(TE_{3-1}, TE_{4-1})	Yes	
DRDF	< r0 / 1 / 0 >	(TE_{3-1}, TE_{4-1})	Yes	2/2 (100%)
	< r1 / 0 / 1 >	(TE_{2-2}, TE_{2-3}),(TE_{4-4},TE_{5-1})	Yes	
WDF	< 0w0 / 1 / - >	(TE_{2-5}, TE_{3-1})	Yes	2/2 (100%)
	< 1w1 / 0 / - >	(TE_{2-1}, TE_{2-2}),(TE_{4-3},TE_{4-4})	Yes	
CFtr	< 0; 0w1 / 0 / - > *a>v*	(TE_{4-2}, TE_{4-4})	Yes	5/8 (62.5%)
	< 0; 0w1 / 0 / - > *a<v*	(TE_{1-1}, TE_{2-2})	Yes	
	< 1; 0w1 / 0 / - > *a>v*	(TE_{1-1}, TE_{2-2})	Yes	
	< 1; 0w1 / 0 / - > *a<v*	(TE_{4-2}, TE_{4-4})	Yes	
	< 0; 1w0 / 1 / - > *a>v*	Not found	No	
	< 0; 1w0 / 1 / - > *a<v*	(TE_{2-4}, TE_{3-1})	Yes	
	<1; 1w0 / 1 / - > *a>v*	Not found	No	
	< 1; 1w0 / 1 / - > *a<v*	Not found	No	
CFdrd	< 0; r0 / 1 / 0 > *a>v*	(TE_{3-1}, TE_{4-1})	Yes	6/8 (75%)
	< 0; r0 / 1 / 0 > *a<v*	(TE_{3-1}, TE_{4-1})	Yes	
	< 1; r0 / 1 / 0 > *a>v*	Not found	No	
	< 1; r0 / 1 / 0 > *a<v*	Not found	No	
	< 0; r1 / 0 / 1> *a>v*	(TE_{4-4}, TE_{5-1})	Yes	
	< 0; r1 / 0 / 1> *a<v*	(TE_{2-2}, TE_{2-3})	Yes	
	< 1; r1 / 0 / 1> *a>v*	(TE_{2-2}, TE_{2-3})	Yes	
	< 1; r1 / 0 / 1> *a<v*	(TE_{4-4}, TE_{5-1})	Yes	
CFwd	< 0 ; 0w0 / 1 /- > *a>v*	(TE_{2-5}, TE_{3-1})	Yes	6/8(75%)
	< 0 ; 0w0 / 1 /- > *a<v*	(TE_{2-5}, TE_{3-1})	Yes	
	< 1 ; 0w0 / 1 /- > *a>v*	Not found	No	
	< 1 ; 0w0 / 1 /- > *a<v*	Not found	No	
	< 0 ; 1w1 / 0 /- > *a>v*	(TE_{4-3}, TE_{4-4})	Yes	
	< 0 ; 1w1 / 0 /- > *a<v*	(TE_{2-1}, TE_{2-2})	Yes	
	< 1 ; 1w1 / 0 /- > *a>v*	(TE_{2-1}, TE_{2-2})	Yes	
	< 1 ; 1w1 / 0 /- > *a<v*	(TE_{4-3}, TE_{4-4})	Yes	

The analyzed March AZ1 algorithm has a slightly lower coverage of CFtr than the March AZ2 algorithm with 14N test complexity. This is because the analysis output presented in Table 3.2 shows that the March AZ1 algorithm cannot detect the CFtr < 1;1w0 / 1 / - >a>v.

When the a-cell is in a specific state, an x' logic can be written to the cell that includes an x logic to sensitize a CFtr occurrence in a v-cell. After the write action, a read operation is performed to look for any improper behavior from the v-cell. One of the test sequences below can be used to sensitize and detect the CFtr < 1; 1w0 / 1 / - >a>v, as per, where F(x) denotes any operation that results in an x-state in the memory cells and * indicates that the related operations are optional

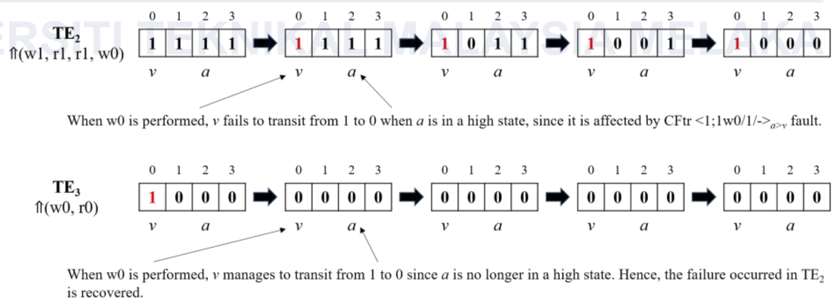
i. **Condition 3.1:** $\Updownarrow$ (…, *F*(1)); $\Uparrow$ (*F*(1)*, w0, w0* r0, *F*(0)*);

ii. **Condition 3.2:** $\Updownarrow$ (…, *F*(1)); $\Uparrow$ (*F*(1)*, w0, w0*); $\Updownarrow$ (r0, …);

iii. **Condition 3.3:** $\Updownarrow$ (…, *F*(1)); $\Updownarrow$ (w0, w0*, r0, *F*(0)*, *F*(1));

The only place in the test sequence for the March AZ1 algorithm where the contents of the cells can change from 1 to 0 is at TE_2: ⇑ (w1, r1, r1, w0), where the w0 operation should set the states of the cells to 0. Therefore, the defective behavior brought on by the CFtr < 1; 1w0 / 1 / - >a>v can be identified by a subsequent read operation. But before carrying out the necessary read operation, that w0 action in TE_2 is followed by another w0 operation in TE_3: ⇑ (w0, r0). As seen in Fig 3.1, which uses a 4-cell memory as an example with the v-cell and a-cell set to address 0 and 2, respectively, the w0 operation in TE_3 really functions as the CFtr <1; 1w0 / 1

/ - >a>v fault recover that masks its occurrence from being detected by the r0 operation in TE_3. Since cell 2, the aggressor cell in TE_2 operation, is in a high state, cell 1, which is impacted by the CFtr <1; 1w0 / 1 /- >a>v fault, is unable to convert its state to low when the w0 operation is carried out. Since the aggressor cell in TE3 is no longer in a high state, the w0 operation in TE_3 manages to successfully convert its state to low.



When w0 is performed, *v* fails to transit from 1 to 0 when *a* is in a high state, since it is affected by CFtr <1;1w0/1/->$_{a>v}$ fault.

When w0 is performed, *v* manages to transit from 1 to 0 since *a* is no longer in a high state. Hence, the failure occurred in TE_2 is recovered.

Figure 3.2 Illustration of the CFtr < 1; 1w0 / 1 / - >$_{a>v}$ fault recovering at TE_3 of the improved March AZ1 algorithm.

To address this, the w0 operation in TE_3 was shifted to the end of TE_2, rearranging TE_2 and TE_3 of the March AZ1 algorithm. Consequently, the test sequence for TE_2 has been adjusted to become $\Uparrow$ (w1, r1, r1, w0, w0) whereas the test sequence for TE_3 has been changed to $\Uparrow$ (r0). As a result, the rearranged TE_2 and TE_3 satisfy the necessary test sequence specified in Condition 3.2 and ought to be capable of identifying the CFtr <1; 1w0 / 1 / - > a>v. With the same 13N complexity and the following test sequence, the recently altered March AZ1 algorithm is now known as the March AZ algorithm: $\Updownarrow$ (w0); $\Downarrow$ (r0, w1); $\Uparrow$ (w1, r1, r1, w0, w0); $\Uparrow$ (r0); $\Uparrow$ (r0, w1, w1, r1); $\Uparrow$ (r1). With the revised March AZ1 algorithm, the fault detection analysis was carried out once more. The CFtr coverage was decreased from 75% to 62.5% but the CFwd was increased from 50% to 75% by the Improved March AZ1, as mentioned in Table 3.2. Additionally, as their coverages are unaltered, it demonstrates that the suggested test element restructuring had no effect on the detections of other FPs.

3.4 Project Development

The implementation of a Memory BIST controller in FPGA using the Improved March AZ1 algorithm was begin with thorough project planning and requirements analysis. During this initial phase, the primary objectives was be established, such as ensuring high fault coverage and efficient implementation on an FPGA. Resources, including appropriate hardware, software, and team expertise, was be allocated, and a detailed timeline with key milestones was be developed. Functional requirements was be clearly defined, specifying the necessary capabilities of the BIST controller, such as test pattern generation and fault detection. Non- functional requirements, like performance benchmarks and area constraints, was also be outlined to guide the development process.

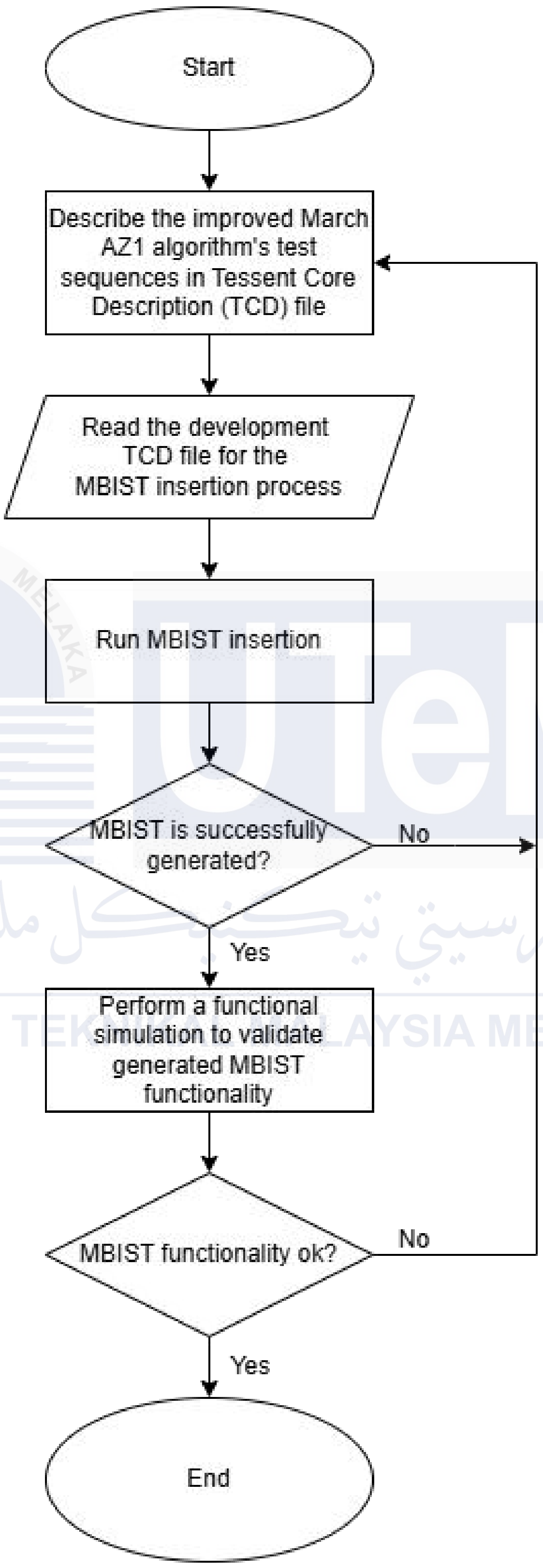

Start
Describe the improved March AZ1 algorithm's test sequences in Tessent Core Description (TCD) file
Read the development TCD file for the MBIST insertion process
Run MBIST insertion
MBIST is successfully generated?
No
Yes
Perform a functional simulation to validate generated MBIST functionality
MBIST functionality ok?
No
Yes
End

Figure 3.3 Flowchart of Implementation of MBIST

The process of Memory BIST (MBIST) insertion was crucial at this stage. MBIST insertion involves generating the MBIST controller and associated circuitry, and it was automated using Tessent MemoryBIST. This tool was streamlined the generation process, ensuring precise and efficient MBIST implementation.

A high-level architecture was generated, including essential components like the test pattern generator, address generator, comparator, control unit, and memory interface. Each component's functionality and interactions was specified, forming a comprehensive blueprint for the HDL coding phase.

HDL coding represented a critical development stage where the high-level design was translated into hardware description language, such as VHDL or Verilog. Each module, from the test pattern generator to the control unit, was meticulously developed and integrated to form a cohesive BIST controller. This phase was involved creating detailed testbenches for simulation, ensuring each module operates correctly in isolation and within the integrated design.

Functional simulations were conducted on a fault-free memory model using the generated MBIST controller. These simulations were checked for any mismatches between the output data and the expected data, help to derive test complexity and ensured that the controller functions correctly under normal conditions. Subsequently, functional simulations were performed on a fault-inserted memory model using the generated MBIST controller to validate fault coverage. These simulations ensured that the MBIST controller can accurately detect and report faults within the memory.

Once the design is verified through simulation, the synthesis and implementation phase was began. The HDL code was synthesized into a gate-level netlist using tools like Altera Quartus, converting the design into a format suitable for FPGA deployment. Timing constraints and pin assignments were defined to ensure the design meets performance criteria and interfaces correctly with the FPGA hardware. The synthesized design was then implemented on the FPGA, followed by placement and routing to optimize its physical layout. This stage were culminated in programming the FPGA with the BIST controller and preparing for hardware testing.

In the final phases, extensive hardware testing and performance evaluation was conducted. The BIST controller was tested in a real-world setup, with the memory under test connected to the FPGA. In-circuit testing was verified the controller's functionality and fault detection accuracy. The same test procedures used in the simulations were applied to the hardware testing. Performance metrics, such as test speed, fault coverage, and resource utilization, were collected and analyzed. It is expected that the hardware testing achieved the same results as the simulations, confirming the reliability and effectiveness of the MBIST controller.

The project was concluded with comprehensive documentation of the design, implementation, and testing processes, along with a user manual and final report. Stakeholder reviews and sign-offs ensured the project meets all objectives and quality standards, culminating in a successful implementation of the Memory BIST controller in FPGA.

3.5 Software and Hardware

3.5.1 Software

There were some software used in this project.

- Questa Sim

The Questa advanced simulator is the core simulation and debug engine of the Questa verification solution; the comprehensive advanced verification platform capable of reducing the risk of validating complex FPGA and SoC designs.

- Tessent Shell MemoryBIST

A tool designed to automate the implementation of Memory Built-In Self-Test (MBIST) controllers. Can ensure a high level of fault coverage, streamline the MBIST implementation process, and enhance the reliability of memory testing in their FPGA or ASIC designs.

- Intel Quartus Design software

Intel Quartus Design software is a comprehensive software tool suite designed for the development of digital logic circuits. It is primarily used for the design, analysis, and implementation of FPGA (Field- Programmable Gate Array) and CPLD (Complex Programmable Logic Device) circuits.

3.5.1 Hardware

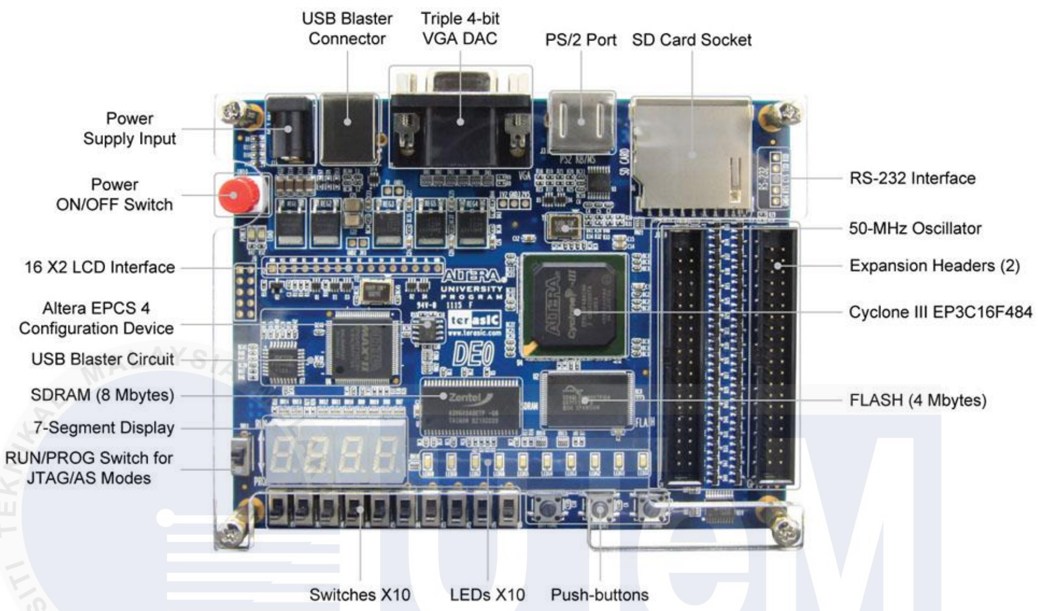


Figure 3.4 FPGAs (Field Programmable Gate Arrays)

Field Programmable Gate Arrays (FPGAs) are versatile integrated circuits widely used across various industries due to their reprogrammable nature.

The FPGA development board to be used in this process is designed to facilitate the implementation and testing of the improved MBIST algorithm within a hardware environment. Typically, such development boards feature a high- capacity FPGA, which provides the reconfigurable logic needed to support complex testing algorithms like MBIST. The board includes multiple I/O interfaces, memory modules, and often integrates tools for programming and debugging, such as JTAG interfaces. The chosen FPGA development board should be compatible with Quartus software, enabling seamless design synthesis, implementation, and functional validation.

In this context, the development board is crucial for verifying the functionality of the modified March AZ1 algorithm in a real-world setting. By implementing the MBIST on the FPGA, developers can test the algorithm's fault detection capabilities and ensure that it performs correctly under various conditions.

The test vectors generator and MBIST circuits were implemented on an Intel Max 10 DE10-Lite FPGA Development Board. The experimental results were captured using the builtin SLA. The onboard SW0 switch, assigned to the *test en* input, was set to high to start the test. It was reset to low once the test was completed and the results were displayed in the SLA window.

3.6 Summary

The methodology for implementing a Memory Built-In Self-Test (BIST) controller in an FPGA using the Improved March AZ1 algorithm involves several key phases. Initially, the project planning and requirements analysis phase sets the foundation. During this stage, clear objectives are defined, resources such as hardware and software tools are allocated, and a detailed timeline with milestones is created. The functional and non-functional requirements of the BIST controller are identified, focusing on performance metrics like fault detection capabilities, speed, and power consumption.

Following the planning phase, the Improved March AZ1 algorithm is studied and optimized. This step includes a thorough analysis of the algorithm's test sequences and fault coverage capabilities to tailor it for efficient hardware implementation. The process of Memory BIST (MBIST) insertion is automated using Siemens Tessent MemoryBIST software, which ensures precise generation of the MBIST controller and associated circuitry.

In the design phase, a high-level architecture of the BIST controller is developed, encompassing components such as the test pattern generator, address generator, comparator, control unit, and memory interface. Each module is described in detail, forming a blueprint for the hardware description language (HDL) coding phase. The design is then translated into HDL, and the modules are rigorously simulated using Siemens ModelSim or QuestaSim to verify their functional correctness.

After successful simulation, the HDL code undergoes synthesis and implementation using Altera Quartus, converting it into a gate-level netlist suitable for FPGA deployment. The design is then implemented on the FPGA, ensuring compliance with timing constraints and pin assignments. Extensive hardware testing is conducted in the final phase, including in-circuit testing to validate the BIST controller's performance. Performance metrics such as test speed, fault coverage, and resource utilization are collected and analyzed. The project concludes with comprehensive documentation and a formal review to ensure the BIST controller meets all specified requirements and functions reliably in its intended environment.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Introduction

This chapter comprehensive reviews about the result of the implementation of Memory BIST Controller in FPGA using the Improved March AZ1 as the Test Algorithm. Section 4.2 mention about the test on the fault free memory model while section 4.3 states about the test on the fault-injected memory mode. Section 4.4 mentions about the test on the Altera Quartus using FPGA and section 4.5 states about the limitation of this project.

4.2 Test on the fault-free memory model

This test was run to evaluate the operation of the MBIST that was put into place and used the March AZ1 and Improved March AZ1 as the UDA. It was assessed by measuring the test completion time, which should equal the UDA's complexity multiplied by N and the duration of the clock used (20 ns), as well as by monitoring the *MBISTPG_GO* flag, which should remain high until the test was finished or until the *MBISTPG_DONE* flag was asserted. Since the test memory was a 1-kB memory, N equaled 1024. The simulation waveform generated by the fault-free memory model test in QuestaSim is shown in Fig.4.1 and Fig 4.2. In this test, the expected data produced by the MBIST controller (*BIST_EXPECT_DATA*) whenever *CMP_E*N is high was compared to the output data read from the memory cell (*dout*). It demonstrates that the test was initiated by asserting the *MBISTPG_GO* flag, which remained high until the test was finished, as shown by a high *MBISTPG_DONE* flag. This observation indicates that there was no difference found during the comparison between dout and *BIST_EXPECT_DATA* when the

CMP_EN signal is high.

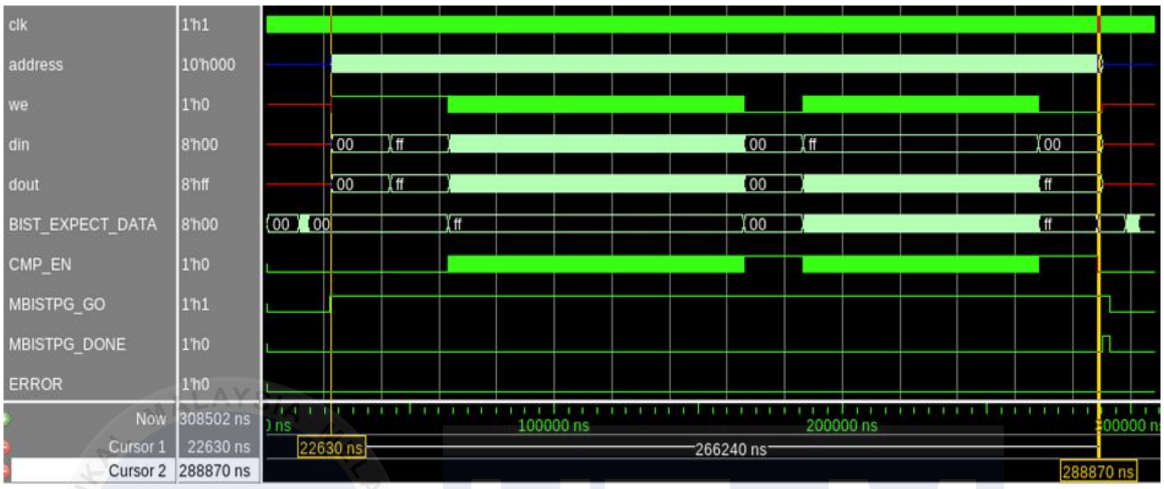


Figure 4.1 The simulation waveform observed in QuestaSim from the test on the fault-free memory model by Improved March AZ1 as the UDA.

For March AZ1 algorithm, 266240ns is the test completion time, measured from the beginning to the conclusion. The test completion time is comparable to what was anticipated because 13 * 1024 * 20ns = 266240ns. As a result, the observation of this test confirmed that the MBIST, which employed the March AZ1 as the UDA, was implemented correctly.

4.3 Test on the fault-injected memory model

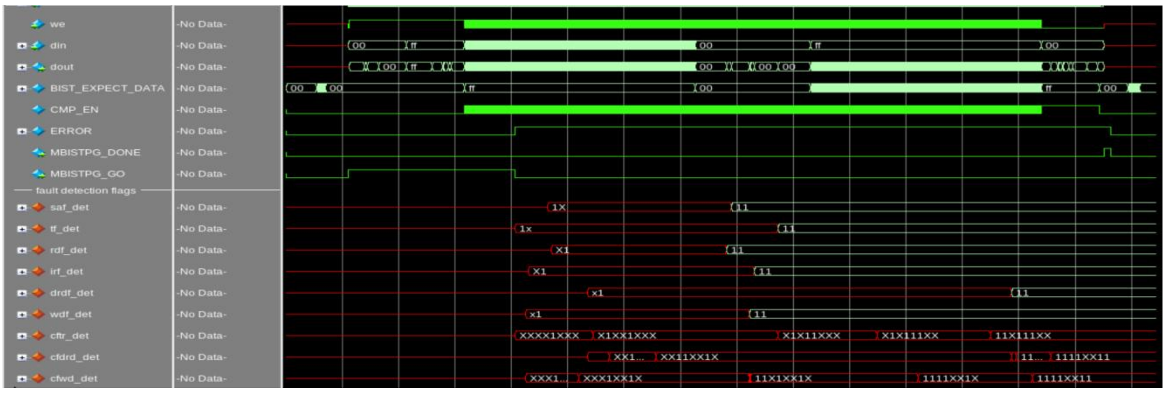


Figure.4.2 The waveform observed from the test on the fault-injected memory, using the Improved March AZ1 as the UDA.

Using the improved March AZ1 as the UDA, Fig.4.3 shows the simulation waveform of the MBIST operation on the fault-injected memory. When the test was finished (shown by a high *MBISTPG_DONE* flag), the values of all fault detection flags were noted and entered into Table 4.1. Consequently, the number of high bits in each fault's detection flag was counted to estimate the fault coverage of the March AZ1 method. As can be seen, the CFtr was reduced and the fault coverage is 5/8 (62.5%) but the CFwd was picked up and increase the fault coverage to 6/8 (75%) by the improved March AZ1 algorithm. Therefore, March AZ1 algorithms have a overall fault coverage of 80.6%.

Table 4.1 The Test Result of the Improved March AZ1 algorithm

Fault Detection Flags	**Corresponding fault**	**Corresponding FP**	**Observed value**	**Derived Fault Coverage (FC)**
saf_detect[1]	*SAF*	*<1/0/->*	*1*	2/2 (100%)
saf_detect[0]		*<0/1/->*	*1*	
tf_detect[1]	*TF*	*<0w1/0/->*	*1*	2/2 (100%)
tf_detect[0]		*<1w0/1/->*	*1*	
rdf_detect[1]	*RDF*	*<r0/1/1>*	*1*	2/2 (100%)
rdf_detect[0]		*<r1/0/0>*	*1*	
irf_detect[1]	*IRF*	*<r0/0/1>*	*1*	2/2 (100%)
irf_detect[0]		*<r1/1/0>*	*1*	
drdf_detect[1]	*DRDF*	*<r0/1/0>*	*1*	2/2 (100%)
drdf_detect[0]		*<r1/0/1>*	*1*	
wdf_detect[1]	*WDF*	*<0w0/1/->*	*1*	2/2 (100%)
wdf_detect[0]		*<1w1/0/->*	*1*	
cftr_detect[7]	*CFtr*	*<0;0w1/0/->a>v*	*1*	5/8(62.5%)
cftr_detect[6]		*<0;0w1/0/->a<v*	*1*	
cftr_detect[5]		*<1;0w1/0/->a>v*	*1*	
cftr_detect[4]		*<1;0w1/0/->a<v*	*1*	
cftr_detect[3]		*<0;1w0/1/->a>v*	*0*	
cftr_detect[2]		*<0;1w0/1/->a<v*	*1*	
cftr_detect[1]		*<1;1w0/1/->a>v*	*0*	
cftr_detect[0]		*<1;1w0/1/->a<v*	*0*	
cfdrd_detect[7]	*CFdrd*	*<0;r0/1/0>a>v*	*1*	6/8(75%)
cfdrd_detect[6]		*<0;r0/1/0>a<v*	*1*	
cfdrd_detect[5]		*<1;r0/1/0>a>v*	*0*	
cfdrd_detect[4]		*<1;r0/1/0>a<v*	*0*	
cfdrd_detect[3]		*<0;r1/0/1>a>v*	*1*	
cfdrd_detect[2]		*<0;r1/0/1>a<v*	*1*	
cfdrd_detect[1]		*<1;r1/0/1>a>v*	*1*	
cfdrd_detect[0]		*<1;r1/0/1>a<v*	*1*	

Fault Detection Flags	**Corresponding fault**	**Corresponding FP**	**Observed value**	**Derived Fault Coverage (FC)**
cfwd_detect[7]	*CFwd*	*<0;0w0/1/->a>v*	*1*	6/8(75%)
cfwd_detect[6]		*<0;0w0/1/->a<v*	*1*	
cfwd_detect[5]		*<1;0w0/1/->a>v*	*0*	
cfwd_detect[4]		*<1;0w0/1/->a<v*	*0*	
cfwd_detect[3]		*<0;1w1/0/->a>v*	*1*	
cfwd_detect[2]		*<0;1w1/0/->a<v*	*1*	
cfwd_detect[1]		*<1;1w1/0/->a>v*	*1*	
cfwd_detect[0]		*<1;1w1/0/->a<v*	*1*	

4.4 Test on the Altera Quartus using FPGA

For Fig 4.5, it shows that the test enable flag is asserted. The signal tap logic analyzer will capture and show it in a waveform. When the test enable is asserted, the MBISTPG_GO signal will start to high and the MBIST will start to operate.

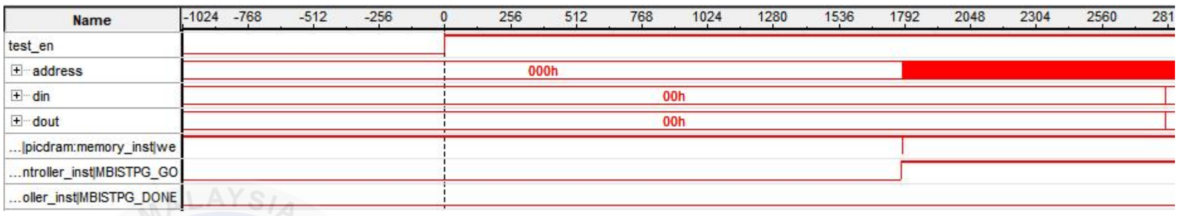


Figure 4.5 Test_en is asserted

For Fig 4.6, the signal tap logic analyzer capture when the MBIST is done. When the test is end, the MBISTPG_DONE signal will start to high and the MBISTPG_GO will remaining to be high to indicates that there were no error between the observed output and expected output. So, it shows that the MBIST that apply the March AZ1 algorithms and improved March AZ1 algorithms have been successfully implemented in FPGA.

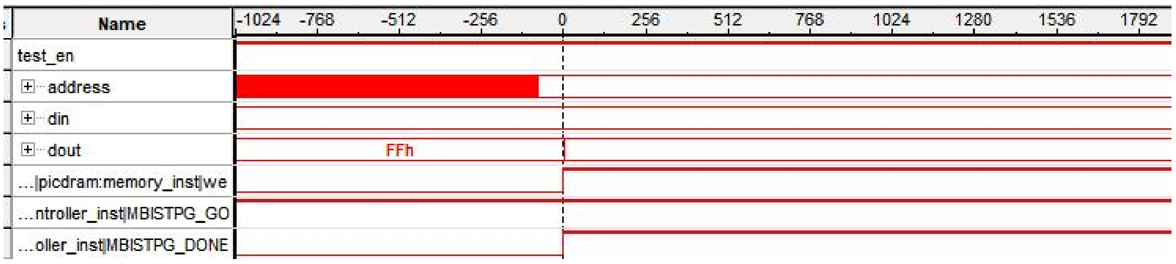
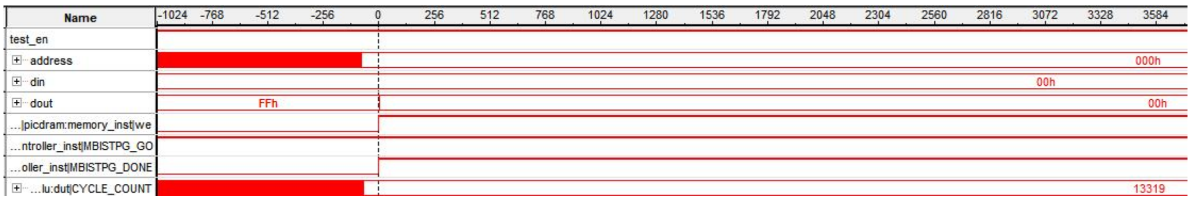


Figure 4.6 Test Done (MBISTPG_DONE = 1)

For Fig 4.7, the CYCLE_COUNT signal has be added to the signal tap analyzer. We can determine the complexity of March AZ1 algorithms and Improved March AZ1 algorithms by observing the cycle count. For March AZ1 algorithms, the cycle count will be around 13*1024 = 13312 due to the 13N complexity and the 1-kB SRAM (N=1024).


Name
test_en
address
din
dout
...|picdram:memory_inst|we
...ntroller_inst|MBISTPG_GO
...oller_inst|MBISTPG_DONE
...lu:dut|CYCLE_COUNT
FFh
000h
00h
00h
13319
Figure 4.7 Cycle count for March AZ1 Algorithms

For Figure 4.8, it shows that the schematic RTL design after the implementation of improved March AZ1 algorithms into FPGA.

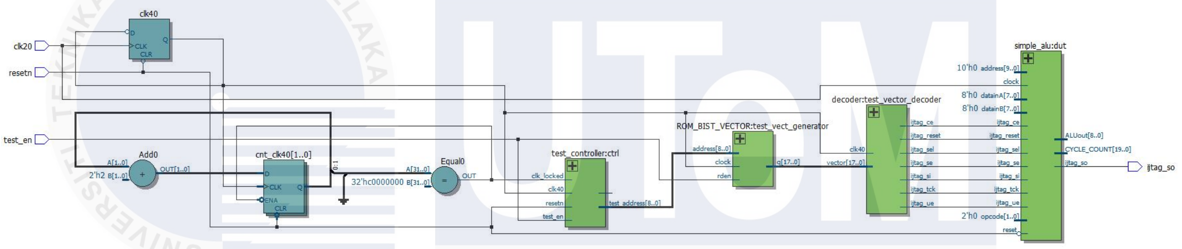
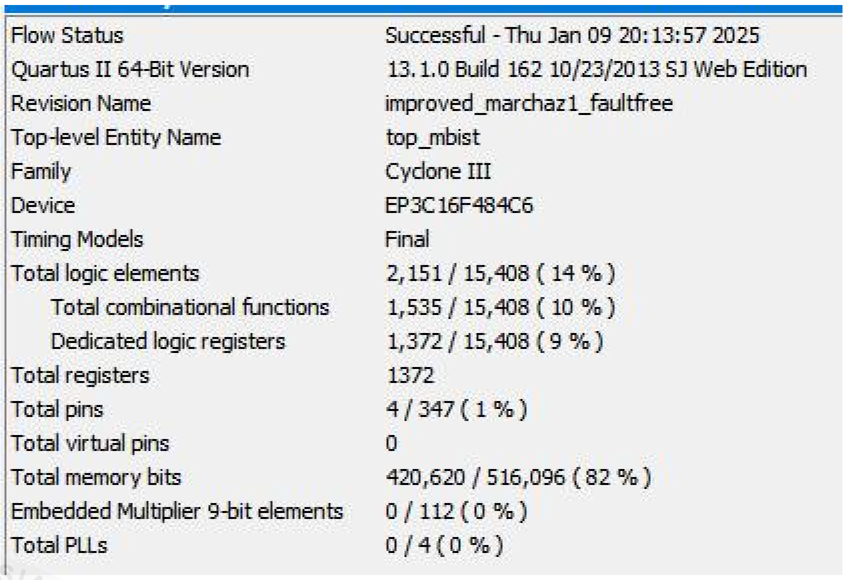


Figure 4.8 Schematic RTL design after implementation into FPGA

For Fig 4.9, it shows that the Altera Quartus version used is 13.0 and thefamily for the FPGA is Cyclone III. In this project, the total registers used is 1372 registers and the total memory bits used is 420620 which is 82% of the total FPGA memory bits. These 82% of memory bits were used in the SRAM, ROM and SignalTap Analyzer.



Flow Status	Successful - Thu Jan 09 20:13:57 2025
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	improved_marchaz1_faultfree
Top-level Entity Name	top_mbist
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	2,151 / 15,408 (14 %)
Total combinational functions	1,535 / 15,408 (10 %)
Dedicated logic registers	1,372 / 15,408 (9 %)
Total registers	1372
Total pins	4 / 347 (1 %)
Total virtual pins	0
Total memory bits	420,620 / 516,096 (82 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 4.9 Complilation Flow Summary

4.5 Limitation

There were some limitation while doing the project. In this project, only fault free models were able to implement into the FPGA using altera quartus software. The fault-injected models were unable to implement into FPGA is due to its Verilog HDL code is not synthesizable. It required more time and effort that not able to done in a short project time to implement the fault-injected models into FPGA.

4.6 Summary

In this project, the result were generated through the Questa simulator and Altera quartus software. For Questa simulator, it help to determine complexity for fault-free improved March AZ1 algorithms and the number of fault detected in the fault-injected improved March AZ1 algorithms. For Altera Quartus software, the Signal tap analyzer helps to compile the output files generated by the Questa simulation software and show the waveform when the test enable is high when the switch on FPGA is asserted. It also show the clock cycle of the improved March AZ1 algorithms which can be used to calculate its complexity.

CHAPTER 5

CONCLUSION

5.1 Conclusion

This project has presented the implementation of the improved March AZ1 algorithm as the UDA in an MBIST controller. The test sequences were described in a TCD file which was used to read and hard-coded into the MBIST controller by the Tessent Shell software during the MBIST insertion process. Simulation were also performed on the generated MBIST controller to validate its functionality, testing time and fault coverage. The improved March AZ1 algorithm remains the complexity of 13N but slightly increases in the overall fault coverage. The simulation of the fault-free model helps to calculate the complexity of the improved March AZ1 algorithm while the simulation of the fault-injected models shows the fault detected for each static fault. Next, this project also implement the improved March AZ1 as the test algorithms into FPGA. This is to test and analyze the improved March AZ1 algorithm in order to assess its fault coverage and complexity.By using the Signal Tap Analyzer, the high input at the test enable and the moment that the MBIST implementation is done can be observed. The Signal Tap Analyzer also shows the clock cycle used for the improved March AZ1 algorithm which used to calculate the complexity of the improved March AZ1 algorithm. In this project, the March AZ1 algorithm has been improved while remaining the complexity lower than March AZ2 algorithm. Next, the improved March AZ1 algorithm was implement into FPGA as a MBIST controller. The complexity and fault coverage of improved March AZ1 algorithm was been evaluate through the test and analysis.

For the first objective, the achievement of developing an improved March AZ1 test algorithm lies in its ability to effectively match the fault coverage of the more complex March AZ2 algorithm, while maintaining a reduced level of complexity. This improvement ensures that the algorithm can identify and address potential faults in memory systems with the same level of precision as the March AZ2, but with greater efficiency and simplicity, thereby offering a more streamlined and practical solution for memory testing. The improved March AZ1 algorithm remains its complexity in 13N while increase in the total fault coverage.

The implementation of a Memory Built-In Self-Test (BIST) controller in Field-Programmable Gate Array (FPGA) using the improved March AZ1 test algorithm marks a significant milestone. This achievement demonstrates the practical applicability of the improved algorithm in real-world hardware, showcasing its compatibility with FPGA technology. By integrating the March AZ1 algorithm into the Memory BIST controller, the system can autonomously test and verify memory integrity, leading to enhanced reliability and performance of memory systems. The improved March AZ1 algorithm is successfully implement into the FPGA board and enable to test the high input of the test enable, high output when the process is done and showing the clock cycle which helps in calculate the complexity.

Evaluating the fault coverage and complexity of the improved March AZ1 algorithm through rigorous tests and analysis has solidified its effectiveness and efficiency. The comprehensive evaluation process has provided detailed insights into the algorithm's capabilities, confirming its ability to detect a wide range of memory faults while maintaining lower complexity. This achievement not only validates the algorithm's performance but also ensures its suitability for various memory testing applications, making it a valuable tool for enhancing the reliability of memory systems.

5.2 Recommendation for Future Work

There were some recommendation for the future work. At first, by further refining the improved March AZ1 test algorithm to enhance its fault coverage while reducing complexity is important. Investigating new techniques and approaches to memory testing can lead to additional breakthroughs, potentially resulting in even more efficient and effective algorithms. This ongoing refinement will ensure that the algorithm continues to evolve and meet the demands of modern memory systems.

Next, the integrating of the improved March AZ1 test algorithm and the Memory BIST controller into more advanced hardware platforms, such as Application-Specific Integrated Circuits (ASICs) and System-on-Chip (SoC) devices, will help assess the algorithm's adaptability and performance across various technological environments. By testing the algorithm in diverse hardware settings, researchers can identify any potential limitations and make necessary adjustments to optimize its performance.

Finally, documenting the research, development, and testing processes of the improved March AZ1 algorithm and Memory BIST controller in detail and sharing this knowledge through publications, conferences, and workshops will contribute to the broader scientific community. This dissemination of information will inspire further research in memory testing and ensure that the advancements made are widely recognized and built upon by future researchers.

5.3 Potential of commercialization

The potential for commercializing the improved March AZ1 test algorithm and Memory BIST controller is promising. Given its enhanced fault coverage, reduced complexity, and successful implementation in FPGA technology, this solution is well-suited for a wide range of industries that rely on reliable memory systems, such as consumer electronics, automotive, aerospace, and telecommunications. The ability to integrate the algorithm into diverse hardware platforms further broadens its market appeal. Companies looking to ensure the integrity and performance of their memory systems will find this technology invaluable, making it an attractive product for commercialization. With continued development and real-world testing, the commercial viability of this innovative memory testing solution is substantial.

REFERENCES

[1] Aiman Zakwan Jidin, Razaidi Hussin, Weng Fook Lee, Mohd Syafiq Mispan, "MBIST Implementation and Evaluation in FPGA Based on Low-Complexity March Algorithms", *Journal of Circuits, Systems and Computers*, vol.33, no.08, 2024.

[2] Jidin, A. Z., Hussin, R., Fook, L. W., Mispan, M. S., Zakaria, N. A., Ying, L. W., & Zamin, N. (2022). Generation of new low-complexity march algorithms for optimum faults detection in SRAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *42*(8), 2738-2751.

[3] Meinerzhagen, P., Teman, A., Giterman, R., Edri, N., Burg, A., Fish, A. (2018). Embedded Memories: Introduction. In: Gain-Cell Embedded DRAMs for Low-Power VLSI Systems-on-Chip. Springer, Cham.

[4] Huq, S. I., Bhuiyan, B. A., & Ahmed, M. (2020). Investigating the Effectiveness of Current Monitoring Methods to Detect Bridging and Stuck-Open Faults in CMOS Circuits. *IOSR Journal of VLSI and Signal Processing*, *10*(4), 01-07.

[5] Aswin, A.M and Ganesh, S.Sankar, Implementation and Validation of Memory Built in Self Test (MBIST) – Survey (September 11, 2019). International Journal of Mechanical Engineering and Technology, 10(3), 2019, pp. 153-160, Available at SSRN: https://ssrn.com/abstract=3451695.

[6] Mukherjee, V., Ghosh, P.K., Maiti, M., Sanyal, J. (2021). Digital Fault Detection Techniques: A Review. In: Das, N.R., Sarkar, S. (eds) Computers and Devices for Communication. CODEC 2019. Lecture Notes in Networks and Systems, vol 147. Springer, Singapore.

[7] Raman, S. R. S. (2024). A Review on Non-Volatile and Volatile Emerging Memory Technologies. *Computer Memory and Data Storage*.

[8] Sharma, A. M., & Naik, J. J. A Review Paper on Memory Fault Models and its Algorithms.

[9] Hantos, G., Flynn, D., & Desmulliez, M. P. (2020). Built-in self-test (BIST) methods for MEMS: A review. *Micromachines*, *12*(1), 40.

[10] Jidin, A. Z., Hussin, R., Fook, L. W., Mispan, M. S., & Loh, W. Y. (2022). Automatic generation of user-defined test algorithm description file for memory BIST implementation. *International Journal of Reconfigurable and Embedded Systems*, *11*(2), 103.

[11] Sahinidis, N. V. (2019). Mixed-integer nonlinear programming 2018. *Optimization and Engineering*, *20*, 301-306.

[12] Dereziński, M., LeJeune, D., Needell, D., & Rebrova, E. (2024). Fine-grained Analysis and Faster Algorithms for Iteratively Solving Linear Systems. *arXiv*

preprint arXiv:2405.05818.

[13] Kong, T., Kim, T., Jeon, J., Choi, J., Lee, Y. C., Park, N., & Kim, S. W. (2022, February). Linear, or non-linear, that is the question!. In *Proceedings of the fifteenth ACM international conference on web search and data mining* (pp. 517-525).

[14] Li, X., Lin, J., & Zhao, L. (2021). Time series clustering in linear time complexity. *Data Mining and Knowledge Discovery*, *35*(6), 2369-2388.

[15] Kumar, A. (2021). Assertion based functional verification of March algorithm based MBIST controller. *arXiv preprint arXiv:2106.11461*.

[16] Singh, K. (2022, June). Performance Analysis of March M & B Algorithms for Memory Built-In Self-Test (BIST). In *2022 IEEE World Conference on Applied Intelligence and Computing (AIC)* (pp. 78-84). IEEE.

[17] Karthy, G., & Sivakumar, P. (2020). Design of Modified March-C Algorithm and Built-In Self-Test Architecture for Memories. *3C Tecnologia*, 219-229

[18] AhmedBasha, S., & Devanna, H. Implementation of Transient Current Testing for Faults in SRAM..

[19] Einfochips, P. E. S. (2021). Memory Testing: MBIST, BIRA & BISR| An Insight into Algorithms and Self Repair Mechanism, Einfochips, dated Dec. 11, 2019, 14 pages. *Retrieved on Oct*, *21.*

[20] Jidin, A. Z., Hussin, R., Mispan, M. S., & Fook, L. W. (2021, November). Novel March test algorithm optimization strategy for improving unlinked faults detection. In *2021 IEEE Asia Pacific Conference on Circuit and Systems (APCCAS)* (pp. 117-120). IEEE..

[21] "IMPLEMENTATION OF MINIMIZED MARCH SR ALGORITHM IN A MEMORY BIST CONTROLLER" by A. Z. Jidin, R. Hussin, M. S. Mispan, W. F. Lee, & N. A. Zakaria. (2023). IMPLEMENTATION OF MINIMIZED MARCH SR ALGORITHM IN A MEMORY BIST CONTROLLER. Journal of Engineering and Technology (JET), 13(2), 67–80.

[22] Jidin, A. Z., Hussin, R., Fook, L. W., Mispan, M. S., Zakaria, N. A., Ying, L. W., & Zamin, N. (2022). Generation of New Low-Complexity March Algorithms for Optimum Faults Detection in SRAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[23] Karthy, G., & Sivakumar, P. (2020). Design of Modified March-C Algorithm and Built-In Self-Test Architecture for Memories. *3C Tecnologia*, 219-229.

[24] OS, N., & K, S. (2021). Architecture for an efficient MBIST using modified march-y algorithms to achieve optimized communication delay and computational speed. *International Journal of Pervasive Computing and Communications*, *17*(1), 135-147.

[25] Kruthika, J., Nisha, G. R., Gayathri, R., & Jeyalakshmi, V. (2022, November). SRAM memory built in self-test using MARCH algorithm. In *2022 International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)* (pp. 1288-1292). IEEE.

[26] Eason, G., Noble, B., & Sneddon, I. N. (1955). On certain integrals of Lipschitz-Hankel type involving products of Bessel functions. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, *247*(935), 529-551.

[27] V. A. Vardanian and Y. Zorian, "A March-based fault location algorithm for static random access memories," in *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, 2002, pp. 256–261, doi: 10.1109/OLT.2002.1030228.

[28] J. H. De Jonge and A. J. Smeulders, "Moving inversions test pattern is thorough, yet speedy," *Comp. Des.*, vol. 1691735, 1976.

[29] S. Hamdioui, Z. Al-Ars, and A. J. Van De Goor, "Testing static and dynamic faults in random access memories," in *Proceedings 20th IEEE VLSI Test Symposium (VTS 2002)*, 2002, pp. 395–400, doi: 10.1109/VTS.2002.1011170.

[30] A. J. Van De Goor, G. N. Gaydadjiev, V. G. Mikitjuk, and V. N. Yarmolik, "March LR: a test for realistic linked faults," in *Proceedings of 14th VLSI Test Symposium*, Apr. 1996, pp. 272–280, doi: 10.1109/VTEST.1996.510868.

[31] S. Hamdioui and A. J. Van De Goor, "An experimental analysis of spot defects in SRAMs: realistic fault models and tests," in *Proceedings of the Ninth Asian Test Symposium*, 2000, pp. 131–138, doi: 10.1109/ATS.2000.893615.

[32] N. A. Zakaria, W. Z. W. Hassan, I. A. Halin, R. M. Sidek, and X. Wen, "Fault detection with optimum March test algorithm," *J. Theor. Appl. Inf. Technol.*, vol. 47, no. 1, pp. 18–27, 2013.

[33] Z. Zhi-chao, H. Li-gang, and W. Wu-chen, "SRAM BIST Design Based on March C+ Algorithm," *Mod. Electron. Tech.*, vol. 34, no. 10, pp. 149–151, 2011.

[34] A. Z. Jidin *et al.*, "Generation of New Low-Complexity March Algorithms for Optimum Faults Detection in SRAM," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 42, no. 8, pp. 2738–2751, 2023, doi: 10.1109/TCAD.2022.3229281.

[35] S. Alnatheer and M. A. Ahmed, "Optimal Method for Test and Repair Memories Using Redundancy Mechanism for SoC," *Micromachines*, vol. 12, no. 7, p. 811, Jul. 2021, doi: 10.3390/mi12070811.

[36] M. A. Ahmed and A. M. Abuagoub, “MBIST Controller Based on March-ee Algorithm,” *J. Circuits, Syst. Comput.*, 2020, doi: 10.1142/S0218126621501607.

[37] G. Harutunyan, V. A. Vardanian, and Y. Zorian, “Minimal march tests for unlinked static faults in random access memories,” in *Proceedings of the IEEE VLSI Test Symposium*, 2005, pp. 53–59, doi: 10.1109/VTS.2005.56.

[38] Z. Cai, Y. Wang, S. Liu, K. Lv, and Z. Wang, “A Novel BIST Algorithm for Low-Voltage SRAM,” in *2019 IEEE International Test Conference in Asia (ITC-Asia)*, 2019, pp. 133–138, doi: 10.1109/ITC-Asia.2019.00036.

[39] Z. Jianping, W. Zhenyu, Y. Jia, P. Wei, and Z. Yun, “An Improved SRAM Fault Built-in-self-test Algorithm,” *J. Hunan Univ. Nat. Sci.*, vol. 46, no. 4, 2019.

[40] S. Hamdioui, A. J. Van De Goor, and M. Rodgers, “March SS: A test for all static simple RAM faults,” in *Records of the IEEE International Workshop on Memory Technology, Design and Testing*, 2002, pp. 95–100, doi: 10.1109/MTDT.2002.1029769

APPENDICES

Appendix 1

Activity	Week																											
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
PSM 1																												
Comprehensive investigation on March algorithms	▨	▨	▨	▨	▨																							
Analyze the original and improved March Az1 algorithm test sequence and fault coverage					▨	▨	▨																					
Plan for the tools and FPGA device for MBIST implementation and validation process							▨	▨	▨	▨	▨	▨																
Prepare PSM 1 report										▨	▨	▨	▨	▨														
PSM 2																												
Synthesis MBIST circuitry design and fit it into targeted FPGA device using Altera Quartus.															▨	▨	▨	▨	▨	▨								
Validate the implemented MBIST through FPGA experimental test.																	▨	▨	▨	▨	▨	▨	▨					
Analyze the implement MBIST performance in terms of its fault coverage, speed, and area																					▨	▨	▨	▨	▨			
Prepare final report																								▨	▨	▨		
Presentation																											▨	▨

Appendix 2

Activity	**Duration (Weeks)**	**Start Week**	**End Week**
PSM 1			
Comprehensive investigation on March algorithms	5	1	5
Analyze the original and improved March Az1 algorithm test sequence and fault coverage	3	5	7
Plan for the tools and FPGA device for MBIST implementation and validation process	6	7	12
Prepare PSM 1 report	5	10	14
PSM 2			
Synthesis MBIST circuitry design and fit it into targeted FPGA device using Altera Quartus.	6	15	20
Validate the implemented MBIST through FPGA experimental test.	7	17	23
Analyze the implement MBIST performance in terms of its fault coverage, speed, and area	5	21	25
Prepare final report	3	24	26

Appendix 3 Improved March AZ1 algorithm TCD file

```
Algorithm (march_AZ1) {
        TestRegisterSetup {
                OperationSetSelect : TessentSyncRamOps;
                AddressGenerator {
                        AddressRegister (A) {
                                LoadColumnAddress: MinColumn;
                                LoadRowAddress: MinRow;
                                X1CarryIn: None;
                                Y1CarryIn: X1CarryOut;
                        }
                }
                DataGenerator {
                        LoadWriteData : 8'b00000000;
                        LoadExpectData : 8'b00000000;
                }
        }
        MicroProgram {
                Instruction (M0_w0){
                         OperationSelect : WriteWriteFastRow;
                         X1AddressCmd : Increment;
                         Y1AddressCmd : Increment;
                         WriteDataCmd : DataReg;
                         InhibitLastAddressCount : on;
 NextConditions {
                                 X1_EndCount : on;
                                 Y1_EndCount : on;
                         }
                }
                Instruction (M1_w1){
                         OperationSelect : WriteWriteFastRow;
                         X1AddressCmd : Decrement;
                         Y1AddressCmd : Decrement;
                         WriteDataCmd : InverseDataReg;
                         InhibitLastAddressCount : on;
                         NextConditions {
                                 X1_EndCount : on;
                                 Y1_EndCount : on;
                         }
                }
                Instruction (M2_w1r1){
                         OperationSelect : WriteRead;
                         ExpectDataCmd : InverseDataReg;
                         WriteDataCmd : InverseDataReg;
                         NextConditions {
                         }
                 }
Instruction (M2_r1w0){
                         OperationSelect : ReadModifyWrite;
                         X1AddressCmd : Increment;
                         Y1AddressCmd : Increment;
                         ExpectDataCmd : InverseDataReg;
                         WriteDataCmd : DataReg;
                         BranchToInstruction : M2_w1r1;
```

```
                    X1_EndCount : on;
                    Y1_EndCount : on;
               }
          }
          Instruction (M3_w0r0){
               OperationSelect : WriteRead;
               X1AddressCmd : Increment;
               Y1AddressCmd : Increment;
               ExpectDataCmd : DataReg;
               WriteDataCmd : DataReg;
               NextConditions {
                    X1_EndCount : on;
                    Y1_EndCount : on;
               }
          }
          Instruction (M4_r0w1){
               OperationSelect : ReadModifyWrite;
               ExpectDataCmd : DataReg;
               WriteDataCmd : InverseDataReg;
               NextConditions {
               }
          }
          Instruction (M4_w1r1){
               OperationSelect : WriteRead;
               X1AddressCmd : Increment;
               Y1AddressCmd : Increment;
               ExpectDataCmd : InverseDataReg;
               WriteDataCmd : InverseDataReg;
               BranchToInstruction : M4_r0w1;
               NextConditions {
                    X1_EndCount : on;
                    Y1_EndCount : on;
               }
          }
          Instruction (M5_r1){
               OperationSelect : ReadReadFastRow;
               X1AddressCmd : Increment;
               Y1AddressCmd : Increment;
               ExpectDataCmd : InverseDataReg;
               NextConditions {
                    X1_EndCount : on;
                    Y1_EndCount : on;
               }
          }
     }
}
```

Appendix 4 TessentSyncRamOps TCD File

```
//
//  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH
//  IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS
//  SUBJECT TO LICENSE TERMS.
//
//  Copyright 1992-2017 Mentor Graphics Corporation
//
//  All Rights Reserved.
//
//  Technology Release: 2018.2
OperationSet(TessentSyncRamOps) {

  PipeliningStages (StrobeDataOut) : 1;

  Operation (NoOperation) { // OP0
    Tick {
      Select       : On;
      OutputEnable : On;
    }
    Tick {
    }
    Tick {
    }
    Tick {
    }
  }

  Operation (Write) { // OP1
    Tick {
      Select                  : On;
WriteEnable           : On;
      ReadEnable              : Off;
      OutputEnable            : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                  : On;
      WriteEnable             : Off;
      ReadEnable              : On;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
    }
  }

  Operation (Read) { // OP2
    Tick {
      Select                  : On;
      WriteEnable             : Off;
      ReadEnable              : On;
```

```
        OutputEnable            : On;
        ShadowReadEnable        : On;
        ConcurrentReadEnable    : On;
        StrobeDataOut;
      }
      Tick {
        Select                  : On;
        WriteEnable             : Off;
        ReadEnable              : On;
        OutputEnable            : On;
        ShadowReadEnable        : On;
        ConcurrentReadEnable    : On;
      }
    }

    Operation (ReadModifyWrite) { // OP3
      Tick{
        Select                  : On;
        WriteEnable             : Off;
        ReadEnable              : On;
        OutputEnable            : On;
        ShadowReadEnable        : On;
        ConcurrentReadEnable    : On;
        StrobeDataOut;
      }
      Tick{
        Select                  : On;
        WriteEnable             : On;
        ReadEnable              : Off;
        ShadowReadAddress       : On;
        ConcurrentReadRowAddress : On;
      }
    }
  Operation (ReadModifyWrite_WithSelectOff) { // OP4
      Tick{
        Select                  : Off;
        WriteEnable             : Off;
        ReadEnable              : On;
        OutputEnable            : On;
        StrobeDataOut;
      }
      Tick{
        Select                  : Off;
        WriteEnable             : On;
        ReadEnable              : Off;
      }
    }
  Operation (WriteReadCompare) { // OP5
      Tick {
        Select                  : On;
        WriteEnable             : On;
        ReadEnable              : Off;
        OutputEnable            : On;
        ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                 : On;
      WriteEnable            : Off;
      ReadEnable             : On;
      OutputEnable           : On ;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
    Tick {
      Select                 : On;
      WriteEnable            : Off;
      ReadEnable             : Off;
      OutputEnable           : On ;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
    }
  }

  Operation (WriteReadCompare_EvenGWE_ON) { // OP6
    Tick {
      Select                 : On;
      WriteEnable            : On;
      OddGroupWriteEnable     : Off;
      EvenGroupWriteEnable    : On;
      ReadEnable             : Off;
      OutputEnable           : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                 : On;
      WriteEnable            : Off;
      OddGroupWriteEnable     : Off;
      EvenGroupWriteEnable    : On;
      ReadEnable             : On;
      OutputEnable           : On;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
 Tick {
      Select                 : On;
      WriteEnable            : Off;
      OddGroupWriteEnable     : Off;
      EvenGroupWriteEnable    : On;
      ReadEnable            : Off;
      OutputEnable           : On;
    }
  }

  Operation (WriteReadCompare_OddGWE_ON) { // OP7
    Tick {
      Select                : On;
      WriteEnable           : On;
      OddGroupWriteEnable     : On;
      EvenGroupWriteEnable    : Off;
      ReadEnable            : Off;
      OutputEnable           : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                : On;
      WriteEnable           : Off;
      OddGroupWriteEnable     : On;
      EvenGroupWriteEnable    : Off;
      ReadEnable            : On;
      OutputEnable           : On;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
    Tick {
      Select                : On;
      WriteEnable           : Off;
      OddGroupWriteEnable     : On;
      EvenGroupWriteEnable    : Off;
      ReadEnable            : Off;
      OutputEnable           : On;
 }
  }

  Operation (WriteReadCompare_AllGWE_OFF) { // OP8
    Tick {
      Select                : On;
      WriteEnable           : On;
      OddGroupWriteEnable     : Off;
      EvenGroupWriteEnable    : Off;
      ReadEnable            : Off;
      OutputEnable           : On;
      ShadowReadEnable        : On;
  ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
```

```
     Tick {
       Select                 : On;
       WriteEnable            : Off;
       OddGroupWriteEnable    : Off;
       EvenGroupWriteEnable   : Off;
       ReadEnable             : On;
       OutputEnable           : On;
       ShadowReadAddress      : Off;
       ConcurrentReadRowAddress : Off;
       StrobeDataOut;
    }
    Tick {
       Select                 : On;
       WriteEnable            : Off;
       OddGroupWriteEnable    : Off;
       EvenGroupWriteEnable   : Off;
       ReadEnable             : Off;
       OutputEnable           : On;
    }
  }

  Operation (Read_WithReadEnableOff) { // OP9
    Tick {
       Select                 : On;
       WriteEnable            : Off;
       ReadEnable             : Off;
       OutputEnable           : On;
       StrobeDataOut;
    }
    Tick {
       Select                 : On;
       WriteEnable            : Off;
       ReadEnable             : Off;
    }
  }
  Operation (ReadModifyWrite_Column_ShadowWriteRead) { // OP10
    Tick{
       Select                     : On;
       WriteEnable                : Off;
   ConcurrentWriteColumnAddress : On;
       ConcurrentWriteDataPolarity  : Inverse;
       ReadEnable                 : On;
       OutputEnable               : On;
       ConcurrentReadEnable       : On;
       StrobeDataOut;
    }
    Tick{
       Select                     : On;
       ConcurrentWriteColumnAddress : Off ;
       WriteEnable                : On;
       ReadEnable                 : Off ;
       ConcurrentReadEnable       : On;
       ConcurrentReadColumnAddress  : On;
    }
  }
  Operation (ReadModifyWrite_Row_ShadowWriteRead) { // OP11
    Tick{
      Select                    : On;
      WriteEnable               : Off;
      ConcurrentWriteRowAddress    : On;
      ConcurrentWriteDataPolarity  : Inverse;
      ReadEnable                : On;
      OutputEnable              : On;
      StrobeDataOut;
    }
    Tick{
      Select                    : On;
      WriteEnable               : On;
      ConcurrentWriteRowAddress    : Off;
      ReadEnable                : Off;
      ConcurrentReadEnable        : On;
      ConcurrentReadRowAddress     : On;
    }
  }

  Operation (WriteRead_Column_ShadowReadWrite) { // OP12
    Tick {
      Select                   : On;
      WriteEnable              : On;
      ReadEnable               : Off;
      OutputEnable             : On ;
      ConcurrentReadEnable       : On;
      ConcurrentReadColumnAddress : On;
    }
    Tick {
      Select                   : On;
      WriteEnable              : Off;
      ConcurrentWriteColumnAddress: On;
      ConcurrentWriteDataPolarity : Inverse;
      ReadEnable               : On;
      OutputEnable             : On;
      ConcurrentReadColumnAddress : Off;
    }
  }
Operation (WriteRead) { // OP13
    Tick {
      Select                 : On;
      WriteEnable            : On;
      ReadEnable             : Off;
      OutputEnable           : On;
      ShadowReadEnable         : On;
      ShadowReadAddress        : On;
      ConcurrentReadEnable     : On;
      ConcurrentReadRowAddress : On;
    }
  Tick {
    Select                : On;
    WriteEnable           : Off;
    ReadEnable            : On;
    ShadowReadAddress     : Off;
    ConcurrentReadRowAddress : Off;
    StrobeDataOut;
  }
}

Operation (ReadRead) { // OP14
  Tick {
    Select                : On;
    WriteEnable           : Off;
    ReadEnable            : On;
    OutputEnable          : On;
    ShadowReadEnable      : On;
    ConcurrentReadEnable  : On;
    StrobeDataOut;
  }
  Tick {
    Select                : On;
    WriteEnable           : Off;
    ReadEnable            : On;
    OutputEnable          : On;
    ShadowReadEnable      : On;
    ConcurrentReadEnable  : On;
    StrobeDataOut;
  }
}

Operation (ReadReadFastRow) { // OP15
  Tick {
    Select                : On;
    WriteEnable           : Off;
    ReadEnable            : On;
    OutputEnable          : On;
    ShadowReadEnable      : On;
    ConcurrentReadEnable  : On;
    RowAddressCount       : On;
    StrobeDataOut;
  }
  Tick {
    Select                : On;
    WriteEnable           : Off;
    ReadEnable            : On;
    OutputEnable          : On;
    ShadowReadEnable      : On;
    ConcurrentReadEnable  : On;
    RowAddressCount       : On;
    StrobeDataOut;
  }
  Tick {
    Select               : On;
    WriteEnable          : Off;
    ReadEnable           : On;
    OutputEnable         : On;
    ShadowReadEnable       : On;
    ConcurrentReadEnable    : On;
    RowAddressCount        : On;
    StrobeDataOut;
  }
}

Operation (ReadReadFastColumn) { // OP16
  Tick {
    Select               : On;
    WriteEnable          : Off;
    ReadEnable           : On;
    OutputEnable         : On;
    ShadowReadEnable       : On;
    ConcurrentReadEnable    : On;
    ColumnAddressCount      : On;
    StrobeDataOut;
  }
  Tick {
    Select               : On;
    WriteEnable          : Off;
    ReadEnable           : On;
    OutputEnable         : On;
    ShadowReadEnable       : On;
    ConcurrentReadEnable    : On;
    ColumnAddressCount      : On;
    StrobeDataOut;
  }
}

Operation (WriteWriteFastRow) { // OP17
  Tick {
    Select               : On;
    WriteEnable          : On;
    ReadEnable           : Off;
    OutputEnable         : On;
    ShadowReadEnable       : On;
    ShadowReadAddress      : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
    RowAddressCount        : On;
  }
  Tick {
    Select               : On;
   WriteEnable          : On;
    ReadEnable           : Off;
    OutputEnable         : On;
    ShadowReadEnable       : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
    RowAddressCount         : On;
  }
  Tick {
    Select                  : On;
    WriteEnable             : On;
    ReadEnable              : Off;
    OutputEnable            : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
    RowAddressCount         : On;
  }
}

Operation (WriteWriteFastColumn) { // OP18
  Tick {
    Select                  : On;
    WriteEnable             : On;
    ReadEnable              : Off;
    OutputEnable            : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
    ColumnAddressCount      : On;
  }
  Tick {
    Select                  : On;
    WriteEnable             : On;
    ReadEnable              : Off;
    OutputEnable            : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
    ColumnAddressCount      : On;
  }
}

Operation (WriteReadWriteInvert) { // OP19
  Tick {
    Select                  : On;
    WriteEnable             : On;
    ReadEnable              : Off;
    OutputEnable            : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
  }
```

```
ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                  : On;
      WriteEnable             : Off;
      ReadEnable              : On;
      OutputEnable            : On;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
    Tick {
      Select                  : On;
      WriteEnable             : On;
      ReadEnable              : Off;
      OutputEnable            : On;
      InvertWriteData         : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
  }

  Operation (ReadWriteReadInvert) { // OP20
    Tick {
      Select                  : On;
      WriteEnable             : Off;
      ReadEnable              : On;
      OutputEnable            : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : Off;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
    Tick {
      Select                  : On;
      WriteEnable             : On;
      ReadEnable              : Off;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                  : On;
      WriteEnable             : Off;
      ReadEnable              : On;
      OutputEnable            : On;
      InvertExpectData        : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : Off;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : Off;
```

```
StrobeDataOut;
    }
  }

  Operation (ReadWriteWriteInvert) { // OP21
    Tick {
      Select                  : On;
      WriteEnable             : Off;
      ReadEnable              : On;
      OutputEnable            : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : Off;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
    Tick {
      Select                  : On;
      WriteEnable             : On;
      ReadEnable              : Off;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                  : On;
      WriteEnable             : On;
      ReadEnable              : Off;
      OutputEnable            : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
      InvertWriteData         : On;
    }
  }

  Operation (WriteWrite) { // OP22
    Tick {
      Select                  : On;
      WriteEnable             : On;
      ReadEnable              : Off;
      OutputEnable            : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable    : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select                  : On;
      WriteEnable             : On;
      ReadEnable              : Off;
```

```
     OutputEnable           : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
  }
}
Operation (ReadAwayReadHome) { // OP23
  Tick {
    Select                 : On;
    WriteEnable            : Off;
    ReadEnable             : On;
    OutputEnable           : On;
    ShadowReadEnable        : On;
    ConcurrentReadEnable    : On;
    StrobeDataOut;
  }
  Tick {
    Select                 : On;
    WriteEnable            : Off;
    ReadEnable             : On;
    OutputEnable           : On;
    SwitchAddressRegister   : On;
    InvertExpectData        : Off;
    ShadowReadEnable        : On;
    ConcurrentReadEnable    : On;
    StrobeDataOut;
  }
}

Operation (WriteWriteInvert) { // OP24
  Tick {
    Select                 : On;
    WriteEnable            : On;
    ReadEnable             : Off;
    OutputEnable           : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
  }
  Tick {
    Select                 : On;
    WriteEnable            : On;
    InvertWriteData         : On;
    ReadEnable             : Off;
    OutputEnable           : On;
    ShadowReadEnable        : On;
    ShadowReadAddress       : On;
    ConcurrentReadEnable    : On;
    ConcurrentReadRowAddress : On;
  }
}
```

```
Operation (WriteAwayReadHome) {  // OP25
    Tick {
     Select               : On;
      WriteEnable           : On;
      ReadEnable            : Off;
      OutputEnable           : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable     : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select               : On;
      WriteEnable           : Off;
      ReadEnable            : On;
      ShadowReadAddress       : Off;
      ConcurrentReadRowAddress : Off;
      SwitchAddressRegister    : On;
      InvertExpectData        : Off;
      StrobeDataOut;
    }
  }

  Operation (ReadWriteRead) { // OP26
    Tick {
      Select               : On;
      WriteEnable           : Off;
      ReadEnable            : On;
      OutputEnable           : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : Off;
      ConcurrentReadEnable     : On;
      ConcurrentReadRowAddress : Off;
      StrobeDataOut;
    }
    Tick {
      Select               : On;
      WriteEnable           : On;
      ReadEnable            : Off;
      ShadowReadEnable        : On;
      ShadowReadAddress       : On;
      ConcurrentReadEnable     : On;
      ConcurrentReadRowAddress : On;
    }
    Tick {
      Select               : On;
      WriteEnable           : Off;
      ReadEnable            : On;
      OutputEnable           : On;
      ShadowReadEnable        : On;
      ShadowReadAddress       : Off;
      ConcurrentReadEnable     : On;
```

```
        ConcurrentReadRowAddress : Off;
        StrobeDataOut;
     }
   }

Operation (WriteReadRead) { // OP27
     Tick {
        Select                  : On;
        WriteEnable             : On;
        ReadEnable              : Off;
        OutputEnable            : On;
        ShadowReadEnable        : On;
        ShadowReadAddress       : On;
        ConcurrentReadEnable    : On;
        ConcurrentReadRowAddress : On;
     }
     Tick {
        Select                  : On;
        WriteEnable             : Off;
        ReadEnable              : On;
        OutputEnable            : On;
        ShadowReadEnable        : On;
        ShadowReadAddress       : Off;
        ConcurrentReadEnable    : On;
        ConcurrentReadRowAddress : Off;
        StrobeDataOut;
     }
     Tick {
        Select                  : On;
        WriteEnable             : Off;
        ReadEnable              : On;
        OutputEnable            : On;
        ShadowReadEnable        : On;
        ShadowReadAddress       : Off;
        ConcurrentReadEnable    : On;
        ConcurrentReadRowAddress : Off;
        StrobeDataOut;
     }
   }




}
```