EMBEDDED MEMORY BIST FOR SYSTEMS-ON-A-CHIP

EMBEDDED MEMORY BIST FOR SYSTEMS-ON-A-CHIP

BY
BAI HONG FANG, B.ENG. (ELECTRICAL)
OCTOBER 2003

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

MASTER OF APPLIED SCIENCE (2003) McMaster University
(Electrical and Computer Engineering) Hamilton, Ontario

TITLE:	Embedded Memory BIST for Systems-on-a-Chip
AUTHOR:	Bai Hong Fang, B.Eng. (Electrical)
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	ix, 89

Abstract

Embedded memories consume an increasing portion of the die area in deep submicron systems-on-a-chip (SOCs). Manufacturing test of embedded memories is an essential step in the SOC production that screens out the defective chips and accelerates the transition from the yield learning phase to the volume production phase of a new manufacturing technology. Built-in self-test (BIST) is establishing itself as an enabling technology that can effectively tackle the SOC test problem. However, unless consciously implemented, its main limitations lie in elevated power dissipation and area overhead, and potential performance penalty and increased testing time, all of which directly influence the cost and quality of manufacturing test. This thesis introduces two new embedded memory BIST architectures, whose objective is to reduce the cost of test and increase the test quality to improve product reliability and yield.

A distributed memory BIST approach with a serial interconnect scheme is first developed. This solution can concurrently support multiple memory test algorithms for heterogeneous memories with low power dissipation during test and with relatively low gate and routing area overhead, in addition to facilitating self-diagnosis. The distributed BIST approach is then extended to a *hardware/software co-testing memory BIST architecture* for complex SOCs. By reusing the existing on-chip resources (e.g., processor cores and busses), further savings in area overhead can be achieved and performance penalty for bus-connected memories can be eliminated. This is accomplished using a design space exploration framework based on a new test scheduling algorithm that balances the usage of the existing on-chip resources and dedicated design for test (DFT) hardware such that the functional power constraints are not exceeded during test, while trading-off the testing time against the DFT area.

Acknowledgments

I will begin by thanking my supervisor, Nicola, for his valuable assistance and energetic support during this project. I also wish to thank my colleagues in the Computer-Aided Design and Test (CADT) Research Group, Qiang Xu, Henry Ko and David Lemstra, who were of great help when ideas and questions needed to be discussed. In particular, I would like to express my appreciation to Qiang Xu for his help in debugging the test scheduling algorithm. I wish to acknowledge Canadian Microelectronics Corporation (CMC) for their manufacturing grants, as well as the technical support and training they have provided. I am also grateful to the graduate students, faculty, administrative and technical members in Department of Electrical and Computer Engineering at McMaster University for their continuous help during my study and research.

Members of my family and many more than I can include here, have loved me far beyond what I can ever return. Special thanks to my uncle Aiguo Chen and aunt Mimi Fang for their steady support over the years. Words cannot express my gratitude to my wife Cizhuang Zhang and my son Wenhan Fang, for their patient understanding, love, and support, which make my work possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to the rapid progress in the very large scale integrated (VLSI) technology, an increasing number of transistors can be fabricated onto a single silicon die. For example, a state-of-the-art 130 nm complementary metal-oxide semiconductor (CMOS) process technology can have up to eight metal layers, poly gate lengths as small as 80 nm and silicon densities of 200K-300K gates/mm^2 [37]. However, although million-gates integrated circuits (ICs) can be manufactured, the increased chip complexity requires robust and sophisticated test methods. Hence, manufacturing test is becoming an enabling technology that can improve the declining manufacturing yield, as well as control the production cost, which is on the rise due to the escalating volume of test data and testing times. Therefore reducing the cost of manufacturing test, while improving the test quality required to achieve higher product reliability and manufacturing yield, has already been established as a key task in VLSI design [8].

1.1 Manufacturing Test of Integrated Circuits

Fabrication anomalies in the IC manufacturing process may cause some circuits to behave erroneously [1]. *Manufacturing test* helps to detect physical defects (e.g., shorts or opens) prior to delivering the packaged circuits to end-users. Once a defective chip has been detected, comprehensive defect screening through *fault diagnosis* is required to adjust the manufacturing process and accelerate the yield learning curve [19].

The physical defects lead to faulty behaviors that can be detected by *parametric tests for chip pins* and *tests for functional blocks* [8]. Parametric tests include DC tests (such as voltage, leakage test and output drive current test) and AC tests (setup and hold time tests and propagation test). These tests are usually technology-dependent and can be done without any understanding of the chip functionality. The test for functional blocks involves modeling manufacturing defects at a certain level of design abstraction, such as behavioral level, register-transfer level (RTL), gate level or transistor level. Fault models based on gate level netlists are technology-independent and over time have been proven to be very efficient for testing digital circuits. Basic fault models for gate level testing are *stuck-at, bridging* and *delay* fault models. The single stuck-at fault model is the most popular fault model in digital system testing and is based on the assumption that a single node (line) in the structural netlist of logic gates can be stuck to a logic value 0 (SA0) or 1 (SA1). The test for functional blocks determines whether the manufactured chip behaves as designed and because the gate count keeps on growing, the testing time for functional blocks is also increasing. Since the time a chip spends on an expensive tester directly influences the production cost, reducing the testing time of functional blocks is an essential task, which needs to be accomplished in order to lower the cost associated with manufacturing test.

The test of functional blocks can further be divided into *structural test* and *functional test*. If the test depends on the netlist structure of the design then it is called *structural test*. Based on the targeted fault models (e.g., stuck-at), automatic test pattern generation (ATPG) tools generate test sets which sensitize the fault and propagate its effects to observation points (e.g., primary outputs). *Functional test* programs, on the other hand, generate a set of test patterns to verify the functionality of each component in the circuit. Because functional test is an exhaustive test method, testing time is prohibitively large for combinational logic blocks, which makes it infeasible for complex digital systems [8]. One exception is the test of semiconductor memories due to their regularity. Since the cells in a memory block have identical structure and they are not related one to each other, and because memory operations are simple (read and write), exhaustive functional test becomes tractable. Chapter 2 gives detailed information on functional memory fault models and test algorithms.

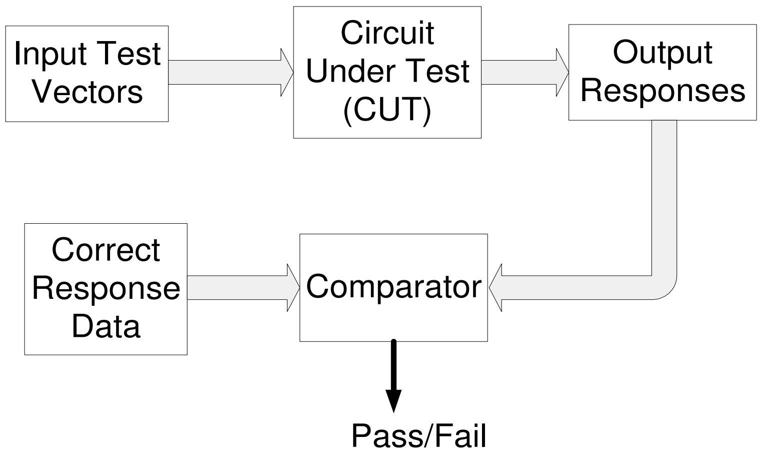


Figure 1.1: Basic Principle of Digital Testing

1.2 Digital Test Methodologies: ATE vs. BIST

The basic principle of manufacturing testing is illustrated in Figure 1.1. *Circuit under test* (CUT) can be the entire chip or only a part of the chip (e.g., a memory core or a logic block). *Input test vectors* are binary patterns applied to the inputs of the CUT and the associated *output responses* are the values observed on the outputs of the CUT. Using a *comparator* output responses are checked against the expected *correct response data*, which is obtained through simulation prior to design tape-out. If all the output responses match the correct response data, the CUT has passed the test and it is labeled as fault-free. Based on the techniques how the test vectors are applied to the CUT and how the output responses are compared, there are two main directions to test electronic circuits: *external testing using automatic test equipment (ATE) and internal testing using built-in self-test (BIST).* When external testing is employed, the input test vectors and correct response data are stored in the ATE memory. Input test vectors are generated using ATPG tools, while correct response data is obtained through circuit simulation. For external testing, the comparison is carried out on the tester. Although the ATE-based test methodology has been dominant in the past, as transistor to pin ratio and circuit operating frequencies continue to increase, there is a growing gap between the ATE capabilities and circuit test requirements (especially in terms of speed and volume of test data).

ATE limitations make BIST technology an attractive alternative to external test for complex chips. BIST [5, 8] is a design-for-test (DFT) method where part of the circuit is used to test the circuit itself (i.e., test vectors are generated and test responses are analyzed on-chip). BIST needs only an inexpensive tester to initialize BIST circuitry and inspect the final results (pass/fail and status bits). However, BIST introduces extra logic, which may induce excessive power in the test mode (see next section for details), in addition to potential performance penalty and area overhead. BIST circuitry can further be divided into logic BIST for random logic blocks (e.g., control circuitry or data path components) and memory BIST for on-chip memory cores. The cost and quality of logic BIST has been subject to extensive research over the last two decades and, since the focus of this thesis is on embedded memory BIST, the reader is referred to [5, 8] for more information. It is important to note that the main problem with logic BIST lies in the computational overhead required to synthesize compact and scalable test pattern generators and response analyzers such that high fault coverage is achieved in low testing time and with limited interaction to external equipment. In contrast, due to the regular memory block structure and simple operations of memory cores, memory BIST (MBIST) can be implemented using compact and scalable test pattern generators and response analyzers and it can rapidly achieve high fault coverage for certain functional fault models (see Chapter 2 for details).

1.3 System-on-a-Chip Test Challenges

As process technologies continue to shrink, designers are able to integrate all or most of the functional components found in a traditional system-on-a-board (SOB) onto a single silicon die, called system-on-a-chip (SOC) . This is achieved by incorporating pre-designed components, known as intellectual property (IP) cores (e.g., processors, memories), into a single chip. While SOCs benefit designers in many aspects, their heterogeneous nature presents unique technical challenges to achieve high quality test, i.e., acceptable fault coverages for the targeted fault models. In the following, several SOC test challenges are enumerated along with the motivation for a shift from

ATE-based SOC testing to BIST.

- *Controllability and observability*

  An SOC contains several embedded IP cores. Although the IP cores are pre-designed and pre-verified by the core providers, SOC composition is the system integrators' duty, who is also in charge of verification and manufacturing testing of the entire SOC, including the IP-protected internal cores. Since most of the input/output (I/O) ports of these embedded cores are not directly connected to the SOC's pins, the *testability* , i.e., both the controllability and the observability [1], is reduced and, unless some special DFT techniques are employed, the fault coverage will be lowered. To increase the testability, test access mechanisms (TAMs) and core wrappers are two new and important DFT techniques in SOC testing [52]. TAM delivers test vectors (propagates test responses) to (from) embedded cores from (to) primary inputs (outputs), while core wrappers (e.g., IEEE P1500 [23, 24, 30]) connect the embedded cores to the TAM. The wrapper/TAM co-design can be solved for different optimization objectives (e.g., testing time or TAM width) and constraints (e.g., layout or power dissipation) [20]. However, when ATE-based testing is employed (i.e., patterns and responses are stored on the tester), since the number of tester channels is limited in practice, test concurrency is bounded by the number of these channels, which can adversely influence the cost of test. This problem can be addressed by moving the generation and analysis functions on-chip and use an inexpensive tester to initialize, control and observe the final results of the testing process.

- *Volume of test data, tester channel capacity and testing time*

  The volume of test data is determined by the chip complexity and it grows rapidly as more IP cores are integrated into a single SOC. The easiest way to deal with increased volume of test data is to upgrade the tester memory and use more tester channels to increase test concurrency, however this is infeasible since it will prohibitively increase the ATE cost. A more cost effective approach is to use test data compaction and/or compression. *Test data compaction* reduces the number of test patterns in the test set (by discarding test patterns that target faults detected by other patterns in the test set) and *test data compression* decreases the number of bits (that need to be stored for each pattern) and uses dedicated decompression hardware (either off or on-chip) for real-time decompression and application [18]. Test data compaction reduces the volume of test data, however it is trading-off the tester channel capacity against the testing time. If the decompression hardware is placed on-chip, then test data compression eliminates this trade-off. *Deterministic BIST* is a particular case of test data compression where the compressed bits are used for BIST initialization (i.e., *seeds*) and BIST observation (i.e., *signatures*). The benefits of memory BIST technology are justified mainly by its deterministic nature.

- *Heterogeneous IP cores*

  Many SOC designs incorporate cores that use different technologies, such as random logic, memory blocks, and analog circuits. For systems assembled on printed circuit boards (PCBs) each of these components was tested using different types of dedicated ATEs (e.g., digital, memory or analog testers). For SOC testing one can use generic high-performance mixed-signal ATEs, however their high production cost brings limited benefits to complex designs, since cores using heterogeneous technologies still need to be tested sequentially, thus lengthening the testing time and ultimately raising the manufacturing test cost. In addition, embedded core controllability and observability issues cannot be addressed without dedicated on-chip DFT hardware, whose necessity justifies a shift toward BIST. The use of different BIST circuitry for the appropriate technologies (logic, memory or analog BIST), increases both testability and test concurrency of SOCs comprising heterogeneous IP cores.

- *At-speed test*

  As VLSI technology moves below 100 nm, traditional stuck-at fault testing is not sufficient. This is because unanticipated process variations, weak bridging defects, and crosstalk violations (only to mention a few) may cause only timing malfunctions, which cannot be detected by the stuck-at fault test vectors delivered by ATEs whose frequency is lower than the maximum CUT frequency.

These logical faults caused by timing-related defects are known as *delay faults* and they can only be detected when the chip is tested at the functional (rated) speed. This type of test is called *at-speed test* [8]. For microprocessor-based circuits, at-speed test can be accomplished by running a set of functional test programs (stored in an off-chip or on-chip memory). Since design automation for functional test program development is still an emerging research area, this approach is very time consuming (even for a decent delay fault coverage). An alternative for logic blocks is to use structural scan patterns and specialized scan chain clocking schemes coupled with two-pattern test application strategies through scan (e.g., broadside or skewed-load [8]). In any of the above cases at-speed test can be performed using high-speed ATEs (note, however, even the highest performance/cost ATEs will be slower than the fastest new chips), or more cost effectively, by BIST interacting with a low-speed testers required only to activate the self-test circuitry and to acquire the BIST signatures.

- *Power dissipation*

  Power dissipation is becoming a key challenge for the deep sub-micron CMOS digital integrated circuits. Placing more and more functions on a silicon die has resulted in higher power/heat densities, which imposes stringent constraints on packaging and thermal management in order to preserve performance and reliability [28]. There are two major sources of power consumption in CMOS VLSI circuits: *dynamic power dissipation*, due to capacitive switching, and *static power dissipation*, due to leakage and subthreshold currents. The 2001 International Technology Roadmap for Semiconductors (ITRS) [19] anticipates that power will be limited more by system level cooling and test constraints than packaging. This is because, if packaging and thermal management parameters (e.g., heat sinks) are determined only based on the functional operating conditions, the higher test switching activity [51] and test concurrency will affect both manufacturing yield and reliability [28].

  On the one hand, *dynamic power dissipation* dominates the chip power consumption for digital CMOS technology in 180 nm range or higher. Dynamic power dissipation can be analyzed from two different perspectives. *Average power dissipation* which stands for the average power utilized over a long period of operation, and *peak power dissipation* which is the power required in a very short time period such as the power consumed immediately after the rising or falling edge of the system clock. When considering SOC test, to achieve high fault coverage with less test data, the test patterns are usually uncorrelated [8]. This means the switching activity during test can differ from that during functional operation. In most cases, the testing power consumption is the higher one. A practical measurement is reported in [34] which indicates the switching activity is 35-40% more during scan-based transition test than that in normal functional mode. For traditional stuck-at fault test, one straightforward solution to meet the power constraints is to reduce the system clock frequency during test which implies longer testing time. However, as described in the previous challenge, to test time related faults, at-speed testing is necessary. Consequently, the power dissipation during at-speed test can exceed the maximum power limit which may lead to chip malfunctions or to burn the overheated chip. There are two research directions to address dynamic power problem during at-speed test: the first direction aims to limit the number of concurrent test blocks using test scheduling under power constraints [12, 13]. The second research direction is to reduce the switching activity during test [11].

On the other hand, *static power dissipation* is becoming an important component for low power design and test in 130nm or lower CMOS technologies with low gate subthreshold. Power gating is an efficient method to reduce static power dissipation and it based on disconnecting the idle module(s) from the power and ground network to reduce the leakage currents. This technique is particularly useful for SOCs with a high number of embedded memories [31]. Note, due to the experimental setup (based on digital 180 nm process technology), the power dissipation problem addressed in this thesis is focused only on dynamic power dissipation during test. However, by using the power-gating method, it is anticipated that the proposed methods can be adapted to solve the static power dissipation problem by turning off the idle memories during test to increase test concurrency.

All the above mentioned SOC test challenges need to be overcome in order to reduce the ever-growing cost of manufacturing test while enabling high manufacturing yield and reliability through satisfactory test quality. Although the cost of test is dominated by many factors, such as the cost of production ATEs, testing time, performance of test automation tools (e.g., ATPG), area and performance overhead caused by additional DFT or BIST circuitry, it is essential to balance this cost against the benefits of enabling high product reliability and a fast yield learning curve. As the SOC complexity increases and more physical defects manifest themselves only in the timing domain, at-speed BIST is emerging as an essential and necessary technology, which can enable short time-to-volume and low cost of manufacturing test. This is also correlated to the fact that, as total chip area continues to increase, the overhead associated with *consciously-designed* BIST architectures is decreasing. The focus of this thesis is to investigate novel cost-effective BIST architectures for embedded memory testing, which is introduced next.

1.4 Embedded Memory Testing

Memory cells are designed using transistors and/or capacitors, and therefore they cannot be modeled by logic gates. Structural test based on gate level netlist cannot be applied to memory testing. However, as mentioned in the previous section, memory cores have a rather regular structure caused by identical memory cells and very simple functional operations (only read and write) which are very suitable for functional test. Unlike the case of random logic testing, which needs a large set of deterministic test patterns to reach the desired fault coverage, functional test programs for embedded memory cores can be generated by compact and scalable on-chip test pattern generators. Furthermore, since written data is unaltered in a fault-free memory, the expected responses can easily be re-generated on-chip and low overhead comparison circuitry can check the correctness of output responses. Therefore, the complexity of memory BIST circuit is lower than that of logic BIST. Due to the deterministic nature and high test quality of memory test algorithms, memory BIST has emerged as the state-of-the-art practice in industry.

Being parts of an SOC, embedded memories face the same test challenges as SOCs. However, the cost of testing embedded memories has unique characteristics and it is influenced by three major components: cost of ATEs, manufacturing testing time, and DFT and BIST area/performance overhead. When considering the challenges faced by SOC testing, reduced testability, high volume of test data, heterogeneous IP cores and at-speed test, can all be solved by implementing programmable embedded memory BIST architectures. However, as tens or even hundreds of heterogeneous memory cores are embedded into a single SOC, power-constrained test scheduling is essential to lower the testing time. In addition, a large number of BISTed memory cores (i.e., memory blocks with BIST circuitry around them) will also induce high routing and gate area overhead, as well as they may adversely influence the memory's speed. Thus, to reduce the overall cost of manufacturing test, it is essential to investigate new memory BIST architectures for complex SOCs, which address the above issues. This is the very purpose of the research work described in this thesis, whose organization and main contributions are summarized in the following section.

1.5 Thesis Organization

New solutions for testing a large number of heterogeneous memories in SOCs are presented in this thesis. The remainder of this thesis is organized as follows. Chapter 2 introduces the memory fault models and summarizes the March test algorithms that use these functional models. Chapter 3 first illustrates the unique challenges faced by memory BIST for large and complex SOCs. This is followed by a comprehensive review of the relevant previous work on embedded memory BIST and test scheduling algorithms. The motivation for *new hardware-centric and hardware/software co-testing memory BIST architectures* is also provided.

Chapter 4 introduces a new *hardware-centric memory BIST architecture* whose objective is to lower the cost of testing heterogeneous memory cores in SOCs that do not comprise programmable processing elements (e.g., microprocessors). The proposed architecture can test all the memories in an SOC, and, to reduce the testing time, it supports partitioned testing with run to completion test scheduling. A detailed hardware implementation of this architecture is provided, followed by experimental results. A more comprehensive solution called *hardware/software co-testing memory BIST architecture* is proposed in Chapter 5. This architecture reuses on-chip resources (e.g., processing units and buses) to test both bus-connected memories and non bus-connected memories in an SOC, which ultimately leads to lower area and performance overhead than previous hardware-centric architecture. The novel hardware and software components of the proposed solution are detailed and a new test scheduling engine tailored for this architecture is described. In the experimental results section, a comparison between hardware-centric, software-centric, and hardware/software co-testing approaches is presented. The trade-off between the testing time and the area overhead explored using the proposed test scheduling engine is also discussed in Chapter 5.

Finally, the conclusion and suggestions for further refinement of the proposed solutions are given in Chapter 6. Appendix A shows the microphoto and the layout view of the two fabricated chips used to empirically validate the correctness of the architectures described in this thesis.

Chapter 2

Theoretical Background on Memory Testing

This chapter introduces the basic theory behind memory testing. There are two kinds of memory test methods: *electrical* (technology-dependent) and *functional* (technology-independent). Electrical memory testing consists of *parametric testing*, which includes testing DC and AC parameters, I_{DDQ} and *dynamic testing* for recovery, retention and imbalance faults [39]. DC and AC parametric tests are used to verify that the device meets its specifications with regard to its electrical characteristics, such as voltage, current, and setup and hold time requirements of chip's pins. Since embedded memories in SOCs usually do not have their I/O ports directly connected to chip's pins, parametric testing for *embedded memories* is not a necessity. I_{DDQ} and *dynamic testing* [25] need a detailed description of the specific process technology. Additional information on electrical testing can be found in [8, 25, 39].

This thesis focuses on technology-independent functional memory testing, whose purpose is to verify the *logical behavior* of a memory core. Because functional memory testing allows for the development of cost-effective short test algorithms (without requiring too much internal knowledge of the memory under test), it is widely accepted by industry as a low-cost/high-quality solution. This chapter provides a theoretical background and explains the memory functional test models and March algorithms. Most of the definitions and figures in this chapter are excerpted from [8, 39].

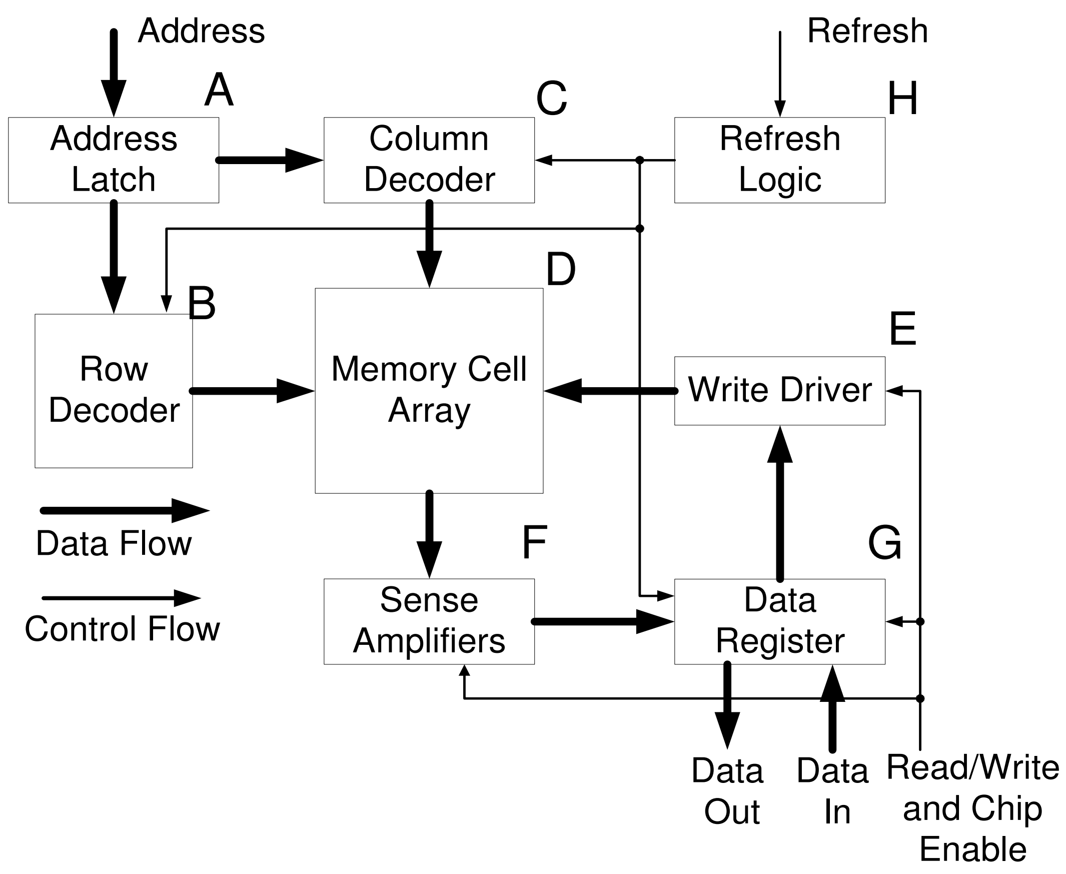


Figure 2.1: Functional Memory Model [39]

2.1 Functional Model and Memory Faults

A *functional model* of a memory is based on its specifications. Figure 2.1 [39] shows the functional model of a dynamic random access memory (DRAM). In this model, the internals of the memory are partly visible, hence it is also referred to as the *gray-box model*. This model can also be reused for modeling faults in synchronous RAM (SRAM), read only memory (ROM) or electrically programmable ROM (EPROM). This can be achieved by adjusting some of the blocks shown in the figure. For example, to model SRAM, one needs to discard the refresh logic block. One of the main advantages of functional models is that they have enough details of data paths and adjacent wires in the memory to adequately model the coupling faults.

	Functional fault		**Functional fault**
a	Cell stuck	i	Address line stuck
b	Driver stuck	j	Open in address line
c	Read/write line stuck	k	Shorts between address lines
d	Chip-select line stuck	l	Open decoder
e	Data line stuck	m	Wrong access
f	Open in data line	n	Multiple accesses
g	Short between data lines	o	Cell can be only set to either 0 or 1
h	Crosstalk between data lines	p	Pattern sensitive interaction between cells

Table 2.1: Subset of Functional Memory Faults [39]

Name	**Functional fault**
SAF	Stuck-at fault
TF	Transition fault
CF	Coupling fault
NPSF	Neighborhood pattern sensitive fault
AF	Address decoder fault

Table 2.2: Reduced Functional Memory Faults [39]

Based on the functional memory model shown in Figure 2.1, a subset of functional memory faults are listed in Table 2.1 [39]. In this table, a *cell* can be either a memory cell or a data register and a *line* is any wiring connection in the memory. In production manufacturing testing once a fault is detected the memory chip is discarded and no diagnosis needs to be undertaken immediately. Failure analysis through fault diagnosis is performed at a later time and more comprehensive test sets (using fault-distinguishing patterns) are applied to identify the source of physical defects. Therefore, for production testing, the faults listed in Table 2.1 [39] can be mapped onto the reduced functional faults shown in Table 2.2 [39]. Table 2.3 [39] summarizes the relationship between the functional faults (Table 2.1) and the reduced functional faults (Table 2.2). For production testing of embedded memories, a great emphasis is placed on March-based test algorithms (see Section 2.3), since they have high defect coverage with a very reasonable hardware cost.

Reduced functional fault		**Functional fault**
SAF	a	Cell stuck
SAF	b	Driver stuck
SAF	c	Read/write line stuck
SAF	d	Chip-select line stuck
SAF	e	Data line stuck
SAF	f	Open in data line
CF	g	Short between data lines
CF	h	Crosstalk between data lines
AF	i	Address line stuck
AF	j	Open in address line
AF	k	Shorts between address lines
AF	l	Open decoder
AF	m	Wrong access
AF	n	Multiple accesses
TF	o	Cells can be only set to either 0 or 1
NPSF	p	Pattern sensitive interaction between cells

Table 2.3: Relationship Between Functional and Reduced Functional Faults[39]

Stuck-at Faults

The *stuck-at fault (SAF)* considers that the logic value of a cell or line is *always 0* (stuck-at 0 or SA0) or *always 1* (stuck-at 1 or SA1). To detect and locate all stuck-at faults, a test must satisfy the following requirement: *from each cell, a 0 and a 1 must be read* [39].

Transition Faults

The *transition fault (TF)* is a special case of the SAF. A cell or line that fails to undergo a $0 \rightarrow 1$ transition after a write operation is said to contain an up transition fault. Similarly, a down transition fault indicates the failure of making a $1 \rightarrow 0$ transition. According to van de Goor [39], a test to detect *and locate* all the transition faults should satisfy the following requirement: *each cell must undergo an $\uparrow$ transition (cell goes from 0 to 1) and a $\downarrow$ transition (cell goes from 1 to 0) and be read after each transition before undergoing any further transitions.*

Coupling Faults

A *coupling fault (CF)* between two cells causes a transition in one cell to force the content of another cell to change. The *2-coupling fault* model [39], which involves only two cells, is defined as follows: a write operation that generates an $\uparrow$ or $\downarrow$ transition in one cell changes the content of the second cell. The 2-coupling fault is a special case of the *k-coupling fault* [39]. A *k-coupling fault* uses the same two cells as the 2-coupling fault, however it allows the fault to occur only when another $k-2$ cells are in a certain state.

- The *inversion coupling fault (CFin)* is a special case of the 2-coupling fault. It means that an $\uparrow$ or $\downarrow$ transition in one cell inverts the content of the second cell. A test to detect all CFins must satisfy the following condition : *for all the cells which are coupled, each cell should be read after a series of possible CFins may have occurred (by writing into the coupling cells), with the condition that the number of transitions in the coupled cells is odd (i.e., the CFins do not mask each other)* [39].

- The *idempotent coupling fault (CFid)* is a another particular case of the 2-coupling fault. It means that an $\uparrow$ or $\downarrow$ transition in one cell forces a second cell to a certain value, 0 or 1. A test to detect all CFids must satisfy the following condition: *for all the cells which are coupled, each cell should be read after a series of possible CFids may have occurred (by writing into the coupling cells), in such a way that the sensitized CFids do not mask each other* [39].

- The *dynamic coupling fault (CFdyn)* is a more general case of the CFid. According to its definition a read or write operation on one cell forces the contents of the second cell either to 0 or 1 [8].

- The *bridging fault (BF)* is caused by a short circuit between two or more cells or lines. It is determined by a logic level rather than a transition write operation. There are two kinds of bridging faults: *AND bridging fault (ABF)*, in which the logic value of the bridge is the AND of the shorted cells or lines, and *OR* *bridging fault (OBF)*, in which the logic value of the bridge is the OR of the shorted cells/lines.

- In the *state coupling fault (SCF)* a coupled cell or line is forced to a certain value (0 or 1) only if the coupling cell is in *a given state*. It is also determined by a logic level.

Neighborhood Pattern Sensitive Faults

A *pattern sensitive fault (PSF)* causes the content of a cell (or the ability to change the content) to be influenced by the contents of other memory cells, which may be either a pattern of 0s and 1s or transitions in memory contents. The PSF is the most general case of the k-coupling fault, where k equals the number of cells in the memory. There are two types of PSF: unrestricted PSF (UPSF) and restricted (or neighborhood) PSF (NPSF) . For tractability reasons, all the known algorithms are tackling the NPSFs, which can be further divided into three types: *active NPSF (ANPSF), passive NPSF (PNPSF), and static NPSF (SNPSF).* NPSF testing algorithms are very complex when compared to March test algorithms [39] (described in Section 2.3). However, for certain process technologies, circuit techniques or memory types, such as high-density DRAMs, testing NPSFs may be a requirement. For further details on NPSFs, the reader is referred to [8, 39].

Address Decoder Faults

Address decoder faults (AFs) represent faults in the combinational logic of the address decoder. Two assumptions are generally accepted: the faults do not introduce sequential behavior in the address decoder and the faults will manifest identically during read and write operations. To simplify the problem, we first consider *bit-oriented memories*, in which only one bit data is stored in each memory location. The March algorithms for testing *word-oriented memories* will be introduced in Section 2.4. The functional faults within the address decoder can be classified into four AFs [39], as shown in Figure 2.2:

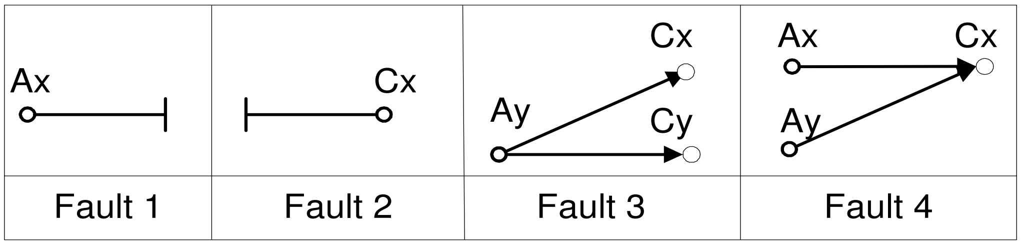


Figure 2.2: Address Decoder Faults [39]

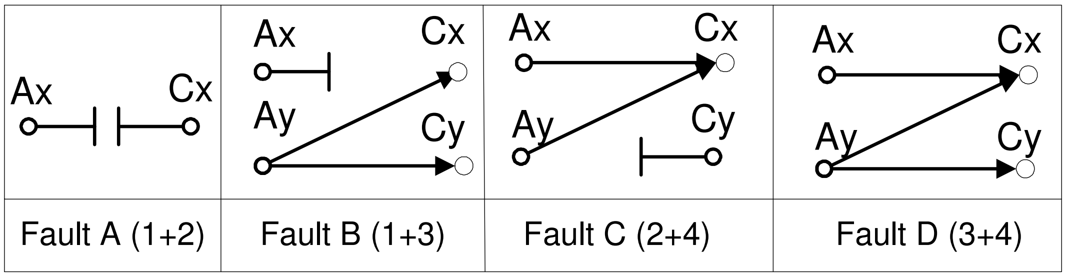


Figure 2.3: Combinations of Address Decoder Faults [39]

- Fault 1: For a certain address, no cell will be accessed.
- Fault 2: A certain cell can never be accessed by any address.
- Fault 3: For a certain address, multiple cells are accessed simultaneously.
- Fault 4: A certain cell can be accessed by multiple addresses.

For bit-oriented memories, because each cell is linked to a dedicated address, none of the faults listed above can stand alone. For example, when fault 1 occurs, then either fault 2 or fault 3 will occur as well. Therefore, in total, four fault combinations in the address decoder are shown in Figure 2.3 [39].

2.2 Fault Combinations

In the previous section we have summarized the most relevant functional fault models and, at the end, we have outlined that in address decoders faults are interrelated. However, when testing a memory core, it is very likely that many various types of

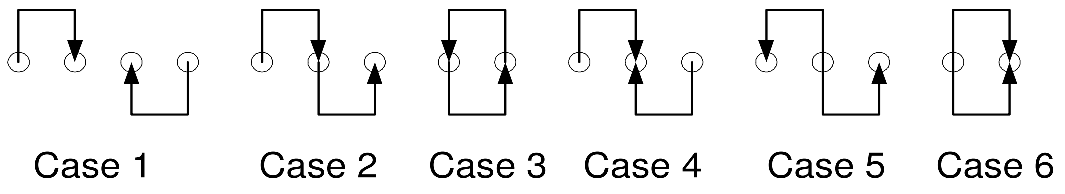


Figure 2.4: Two Coupling Faults [39]

faults may occur simultaneously. These faults can be *linked* or *unlinked*. In a linked fault one fault may influence the behavior of other faults. An unlinked fault does not influence the behavior of other faults. Linked faults can be further classified as linked with the same fault type or linked with different fault types.

Linked Faults of the Same Fault Type

Since a SAF involves only one cell and only one SAF can occur in a single cell, a SAF cannot be linked with another SAF. Similarly, a TF also cannot be linked with another TF. In a CF, two cells are involved. Figure 2.4 [8] lists 6 different cases when two coupling faults (4 cells involved) occur concurrently. Case 1,2,3, and 5 are unlinked faults because each of the coupled cells are only coupled in only one way. Case 4 is a linked fault because a cell is coupled to more than one cell and case 6 is also a linked fault because a cell is coupled to a single cell in more than one way. As stated in [39], in general, linked CFins cannot be detected by March tests. However, tests for idempotent CFs (CFids) will detect inversion CFs (CFins).

Linked Faults of Different Fault Types

When a test for a certain fault type is performed, it will cover faults at a lower hierarchical level (see Table 2.2) [39]. When SAFs link with TFs and/or CFs, they can be detected without any extra tests for TFs and/or CFs. For unlinked TFs and CFs, testing CFs can also detect TFs. However, if TFs are linked with CFs then require a new test. This is because a TF may mask a CF, while the CF masks the TF. For a full description of linked faults, the reader is referred to [8, 39].

2.3 Functional Testing and March Test Algorithms

Based on the used memory fault models, memory test algorithms can be divided into four categories [39] as described below:

1. Traditional tests including *Zero-One, Checkboard, GALPAT and Walking 1/0, Sliding Diagonal, and Butterfly [39].* They are not based on any particular functional fault models and over time have been replaced by improved test algorithms, which result in higher fault coverage and equal or shorter test time.
2. Tests for stuck-at, transition, and coupling faults that are based on the reduced functional fault model and are called March test algorithms [39].
3. Tests for neighborhood pattern sensitive faults.
4. Other memory tests: any tests which are not based on the functional fault model are grouped in this category.

As mentioned in Section 2.1, March test algorithms can efficiently test embedded memories and, therefore, the rest of this section provides more details about them.

March Test Notation

A *March test* consists of a finite sequence of March elements [39]. A *March element* is a finite sequence of *operations or primitives* applied to every memory cell before proceeding to next cell [39]. For example, $\Downarrow (r1, w0)$ is a March element and $r0$ is a March primitive. The address order in a March element can be increasing ($\Uparrow$), decreasing ($\Downarrow$), or either increasing or decreasing ($\Updownarrow$). An *operation* can be either writing a 0 or 1 into a cell ($w0$ or $w1$), or reading a 0 or 1 from a cell ($r0$ or $r1$). In summary, the notation of March test is described as follows:

$\Updownarrow$	Addressing order can be either increasing or decreasing;
$\Uparrow$	Increasing memory addressing order;
$\Downarrow$	Decreasing memory addressing order;

Name	Algorithm
MATS	$\{\Updownarrow (w0); \Updownarrow (r0, w1); \Updownarrow (r1)\}$
MATS+	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$
MATS++	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0)\}$
MARCH X	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$
MATCH C-	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$
MATCH A	$\{\Updownarrow (w0); \Uparrow (r0, w1, w0, w1);$ $\Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$
MATCH Y	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Downarrow (r1, w0, r0); \Updownarrow (r0)\}$
MATCH B	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1);$ $\Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$

Table 2.4: Irredundant March Test Algorithms [8]

r0	Read 0 from a memory location;
r1	Read 1 from a memory location;
w0	Write 0 to a memory location;
w1	Write 1 to a memory location;

March Test Algorithms

Table 2.4 [8] lists several relevant March algorithms reported in the literature. Table 2.5 [8] gives the fault coverage and the operation count of these March algorithms, which are also called *irredundant algorithms* (by removing any operation from the test, the targeted fault coverage will be reduced). To generate custom March algorithms for improved defect coverage in new process technologies, an effective methodology was proposed in [2].

March algorithms are very easy to implement in either software or hardware. A piece of pseudo-code for the MATS+ algorithm is given to demonstrate the basic test procedures. In the code shown below, n is the total number of bits of the memory (bit-oriented memory) and *Addr[i]* points to the i_{th} memory address for read or write. Line 1 runs the first March element of MATS+ algorithm $\Updownarrow (w0)$. Since the address sequence can be either up or down, here we use an up address sequence. Line 2 to 5 run the second March element $\Uparrow (r0, w1)$ with the up address sequence. Line 6 to

	Fault Coverage								
Algorithm	**SAF**	**AF**	**TF**	**CF in**	**CF id**	**CF dyn**	**SCF**	**Linked Faults**	**Oper. Count**
MATS	All	Some							4.n
MATS+	All	All							5.n
MATS++	All	All	All						6.n
MARCH X	All	All	All	All					6.n
MARCH C-	All	All	All	All	All	All	All		10.n
MARCH A	All	All	All	All				All linked CFids, some CFins linked with CFids	15.n
MARCH Y	All	All	All	All				All TFs linked with CFins	8.n
MARCH B	All	All	All	All				All linked CFids, all TFs linked with CFids or CFins, some CFins linked with CFids	17.n

Table 2.5: Irredundant March Test Summary [39]

9 implement the third March element $\Downarrow (r1, w0)$ with the down address sequence. If any data mismatch happened during the test (line 3 and 7) the program will stop and return fail. Otherwise, it will return success after all March elements are finished.

MATS+ Test

```
1. for (i = 0; i < n-1; i++) Addr[i] = 0;
2. for (i = 0; i < n-1; i++) {
3.     if (Addr[i] != 0) return (fail);
4.     Addr[i] = 1;
5.   }
6. for (i = n-1; i >= 0; i– –) {
7.     if (Addr[i] != 1) return (fail);
8.    Addr[i] = 0;
9.   }
10. return (success);
```

Characteristics of March Algorithms

March-based memory test algorithms have several important characteristics:

- Up (down) address sequence must be the exact reverse down (up) sequence, however its internal order is irrelevant. For example, if a 3 bits up address sequence is $\{0, 5, 2, 3, 7, 1, 4, 6\}$, then the down sequence must be $\{6, 4, 1, 7, 3, 2, 5, 0\}$.
- Most March algorithms are only a simple combination of several March elements (e.g., $\Uparrow (r0, w1)$ is a March element). By analyzing the March algorithms shown in Table 2.5, it can be observed that the background pattern during execution can be inferred by the previous operation. For example, a read operation infers the same background data used in the last operation. Similarly, a write operation infers the reversed background data used in the last operation. For example, the first operation of the March C- test shown in Table 2.4 is *w0*. The next operation is read (must be *r0*) and the following operation is write (must be *w1*). Based on this observation, one can reduce the number of March elements and the complexity of their implementation. For Match C-, only three March elements are needed: *(w), (r, w), (r)*. The total number of March elements for the most practical March algorithms is less than ten.
- Using the test generation method proposed in [2], one can generate novel March algorithms (based on a limited number of March elements implemented in hardware), to detect new technology-specific faults.
- For word-oriented memories, one needs to run the March test several times using different background patterns [39, 47] to improve the fault coverage or to use modified March algorithms, such as March-CW [46], to reduce the testing time. Table 2.6 gives an example of background patterns for a 8-bit word memory.

March Algorithms with Diagnosis and Repair Support

When a memory is fabricated using new technology, it is desirable to have a fast yield learning curve[19]. Therefore, it is critical to perform very detailed failure

	Normal	**Inverse**
1	00000000	11111111
2	01010101	10101010
3	00110011	11001100
4	00001111	11110000

Table 2.6: Background Patterns for an 8-bit Memory Width

analysis through fault diagnosis to identify the particular defects (for example, it is essential to distinguish faults between SAF and CF). The ultimate outcome of failure analysis is a redesigned set of masks for the next fabrication run, which will improve the manufacturing yield. A new set of March algorithms will be used for the best process-specific fault coverage. For large memory chips or SOCs with large embedded SRAMs or DRAMs, to increase the yield, it is crucial to also use redundant memory locations to repair the faulty rows (columns) [53]. This leads to new type of algorithms, called fault location algorithms. This type of algorithms can, for example, locate the aggressor cell of a coupling fault (CF).

A complete solution targeting fault diagnosis and fault location has three components: a memory BIST architecture with diagnosis support to save and send out the diagnostic information, a diagnostic test algorithm and a tool to analyze the collected diagnostic data and generate a detailed fault report for failure analysis and a fault bitmap for repair purposes. The traditional March test algorithms (shown in Table 2.5) are aimed at detecting faults and they do not support implicitly fault diagnosis and fault location. To address this problem, several diagnostic March tests were introduced in [48] to distinguish the traditional reduced faults listed in Table 2.2. In [6], both fault diagnosis and fault location algorithms were analyzed. To efficiently address fault location problem, a March-based fault location algorithm was proposed also in [40]. Since all of these recently proposed algorithms are March-based, they have the same characteristics as the traditional March test algorithms introduced in this section and can be supported by the existing March-based memory BIST architectures. In the following chapter, the relevant previous work on memory BIST architectures is described and the motivation for the proposed solutions is given.

Chapter 3

Previous Work on Memory BIST and Motivation

The previous chapter has introduced the basic fault models and test algorithms for semiconductor memories. If the memories are embedded into an SOC (i.e., chip's I/Os are not directly connected to the memories' ports) then how are the test vectors applied and how are the test responses observed? As introduced in Chapter 1, there are two main approaches for testing embedded memories: external test by direct access using ATE and internal test using BIST. On the one hand, direct access to the embedded memory cores from the limited number of I/O pins needs a high-performance ATE, as well as very long testing time since tester channels are time-shared by different memories under test. Thus, external test becomes infeasible, in particular for large SOC devices where transistor to pin ratio is high. On the other hand, BIST provides at-speed and high-bandwidth access to the embedded memory cores, and it only needs a low cost ATE to initialize the test sessions and to inspect the final results. However, although BIST is state-of-the-art technology for embedded memory testing, unless carefully designed, it may induce excessive power, in addition to performance and area overhead. Since embedded memories account for more than 60% of the silicon area in modern SOCs [33] (up to 95% by 2016 according to [19]) this chapter describes the relevant approaches to embedded memory BIST, summarizes their strengths and limitations and motivates the research presented in this thesis.

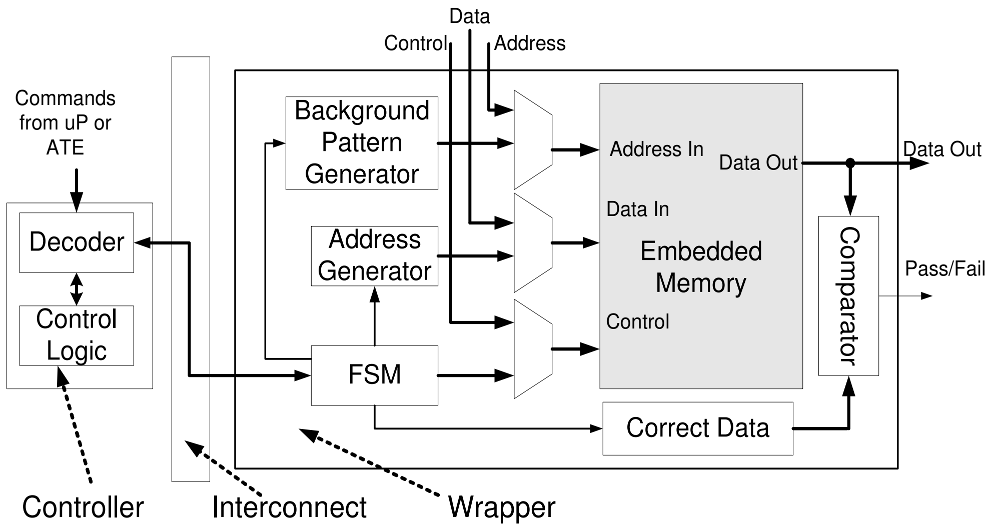


Figure 3.1: Generic Memory BIST Architecture

3.1 Memory BIST Challenges

A typical embedded memory BIST (MBIST) approach comprises an MBIST wrapper, an MBIST controller and the interconnect between them, as shown in Figure 3.1. The MBIST wrapper further includes *an address generator* to provide *complete* memory address sequences (i.e., for n address lines all the 2^n locations are visited in a complete sequence); *a background pattern generator* to produce data patterns when testing word-oriented memories (as described in the preceding chapter); *a comparator* to check the memory output against the expected correct data pattern; and *a finite state machine (FSM)* to generate proper test control signals based on the commands received from the MBIST controller. The MBIST controller pre-processes the commands received from upper-level controller (either on-chip microprocessor or off-chip ATE) and then sends them to the MBIST wrapper. The interconnect between the wrapper and the controller could be either serial (i.e., a single command line is shared by all the wrappers) or parallel (i.e., dedicated multiple command lines are linking different wrappers to the controller). Note, the previously described partition of the MBIST architecture and the terms 'MBIST wrapper' and 'MBIST controller' are not universal, and only applicable in this thesis.

BIST addresses most of the challenges faced by testing embedded memories in an SOC (see Chapter 1 for a full description of SOC testing challenges). However, the increasing size *and* number of embedded memory cores and the rapid development in VLSI process technologies lead to unique requirements for embedded memory BIST.

1. **Support multiple test algorithms**: The conventional MBIST approaches usually implement a single March test algorithm. However, deep submicron process technologies and design rules introduce physical defects that are not screened when using the memory test algorithms developed for previous process generations. Therefore MBIST architectures should be programmable to support multiple memory test algorithms to increase the fault coverage and to find the most suitable algorithms for the manufacturing process at hand.

2. **Diagnosis and repair support**: Diagnosis support in an MBIST architecture is mandatory for manufacturing yield enhancement for new process technology and a rapid transition from the yield ramp phase to the volume production phase [19]. Furthermore, since embedded memories are subject to more aggressive design rules, they are more prone to manufacturing defects (caused by process variations) than other cores in an SOC. For large embedded memory cores, the manufacturing yield can be unacceptable low (e.g., for a 24Mbits memory core, the yield is around 20% [53]). Hence, to achieve a certain manufacturing yield, in addition to diagnosis support, it is also beneficial to introduce self-repair features comprising redundant memory cells.

3. **Test heterogeneous memories**: State-of-the-art SOCs include many types of memory cores, such as, among others, SRAM, DRAM, flash and ROM. Traditional MBIST approaches were designed to test only one type of memory. However, to reduce area and routing overhead via hardware resource sharing, as well as to decrease the testing time, it is advantageous to develop MBIST architectures that support testing heterogeneous memories simultaneously.

4. **Power dissipation constraints**: As introduced in Chapter 1, more power dissipation is expected during test mode than power consumed during normal

functional mode for scan-based SOC testing. However, because memory test is functional test, for each memory the power dissipation will be identical in both test mode and normal functional mode. Therefore, if all memory blocks in an SOC can be activated simultaneously during functional mode, power dissipation will not exceed the maximum power constraint during test. Hence, no test scheduling is required in this case. However, to reduce the overall testing time, test scheduling is still necessary for memory testing as described in the following.

On the one hand, for bus-connected memories (BCMs) which are connected to a single-master bus architecture [4], only one BCM can be accessed at any time during functional mode. If all BCMs are wrapped, then all of them can be activated simultaneously during test. Consequently, the power dissipation will be higher during test than during functional operation, and therefore, test scheduling is necessary.

On the other hand, memory testing is part of SOC testing. It was proven in [34] that cores which use scan-based test methodology will consume more power during test than during functional mode. If the testing time of these scan-based cores is longer than that of memory cores, then by relaxing the power constraints for scan-based core testing and carefully scheduling memory testing with tightened power constraints, the overall testing time for the SOC can be reduced.

Since test scheduling under power constraints is highly interrelated to the resource sharing mechanisms used in the MBIST architecture, it is essential to develop new power-constrained test scheduling algorithms that will get the maximum usage of the available hardware resources for embedded memory testing.

5. **Reuse the available on-chip processing/communication resources**: SOCs usually contain one or more processing elements (e.g., microprocessors), which use on-chip system busses to communicate with other cores. Hence the embedded memory cores in an SOC can be divided into two groups: *bus-connected memories (BCMs) and non bus-connected memories (NBCMs)* . Although all the embedded memory cores can be tested by adding dedicated memory BIST

wrappers, the high area overhead of BIST circuitry, as well as the performance penalty caused by intrusive DFT hardware may prove to be the main drawback of this approach. Therefore, reusing the available on-chip resources for testing the embedded memories can lower the area and performance overhead associated with a high number of dedicated MBIST wrappers for BCMs. Furthermore, by implementing non-time-critical tasks in software using a processor, the complexity of the controller can also be reduced.

6. **Design reuse**: Reusing IP cores in an SOC can greatly simplify the design phase and cut down the time to market. The Reuse Methodology Manual [21] lists various features to make a core reusable. A reusable MBIST core with a scalable and portable architecture, associated with a clear methodology for design flow integration, can significantly reduce the cost of test preparation.

The objective of memory BIST approaches is to meet some or all of the above requirements while reducing the cost of test by targeting low area and performance penalty and low testing time. The existing approaches have explored three main directions to gain improvements: *memory BIST architectures, test scheduling algorithms, and special design implementations.* Due to their interrelation, without a good architectural support it is hardly possible to achieve any significant improvements through test scheduling or special design techniques. The following sections will review the relevant MBIST approaches presented in the literature.

3.2 Memory BIST Architectures

A memory BIST architecture is defined by the integration of its three components shown in Figure 3.1 (controller, wrapper and interconnect). A *standalone* approach uses a dedicated wrapper and controller for each memory core (or memory cluster with several identical memory cores), while a *distributed* approach shares one controller to manage some or all of the MBIST wrappers in an SOC.

Standalone MBIST Architecture

In a standalone MBIST architecture , the BIST controller and the wrapper are physically close located, hence parallel interconnect between them can be used. The MBIST approach of each memory is independent of the other memories' BIST approaches, which makes the implementation of this approach straightforward. However, based on the specific test requirements of different memories and technologies, it needs to be improved in one or more aspects, as described in the following.

MBIST approaches which support multiple March test algorithms are called programmable MBIST architectures . Based on the structure of March algorithms provided in Chapter 2, to support multiple March test algorithms, one can either implement all the March primitives or several March elements. Since there are only four *March primitives* ($r0$, $w0$, $r1$, $w1$), by implementing all of them with different combinations of background patterns and address sequences, any March algorithm can be supported. One programmable MBIST approach using March primitives was investigated in [50] and it includes an instruction memory to store the test instructions and a decoding logic to process the test instructions. *March element*-based approaches implement only several most commonly used March elements. Based on the implemented March elements, only a limited number of March algorithms can be supported. However, its main advantage lies in less area overhead (simpler decoding logic and less test instructions) when compared to March primitive-based approaches. In addition, by carefully selecting the March elements, new March test algorithms can be generated [2] to target memory faults in new process technologies. A programmable FSM-based MBIST architecture with 7 March elements was researched in [50]. Another March element-based approach, which supports 40 March algorithms, was presented in [47]. However, both approaches use dedicated on-chip memory to store the test instructions, thus leading to large test area overhead. Furthermore, dedicated control signals are needed for each MBIST core, which may cause routing and test integration problems when the SOC comprises hundreds of memory cores. To overcome the control problem, a P1500-based [30] programmable MBIST architecture using March elements was introduced in [22]. Using P1500 core wrappers, the test controller (ATE or on-chip processing element) has the full controllability of all the wrapped memories and can send different test instructions to each MBIST wrapper during the test, thus eliminating also the need of an on-chip instruction memory. Note, however, if the SOC consists of a high number of embedded memory cores, and all of them are wrapped with fully-compliant P1500 wrappers, the main limitation is caused by the excessive wrapper area overhead and unnecessary performance degradation.

Diagnosis support is another important feature of MBIST architectures. A built-in self-diagnosis (BISD) scheme was introduced in [46]. It sends out faulty memory cell information (such as faulty address, data, and test session number) for failure analysis. To reduce the control complexity of this approach when testing numerous memory cores, a P1500 MBIST approach with diagnosis enhancement was proposed in [3]. To reduce the testing time in the diagnosis mode (caused by the serial scan-chain structure required to shift out the diagnosis information), a test response compression method was introduced in [10]. Using this method, less I/O pins can be used to send out the faulty response data compared with the uncompressed parallel solution. Due to the increased size of embedded memories, support for memory self-repair is becoming necessary to increase the overall SOC yield. Using the detailed location and information of faulty memory cells (provided by diagnosis support approaches discussed above), one can perform memory redundancy allocation and use fuse-boxes or other methods to repair the faulty memories. However, to collect enough information on fault locations for various memory faults, more complex March test algorithms are required, which implies longer testing time. An MBIST solution was introduced in [15] to test and repair large embedded DRAMs using on-chip redundancy allocation. To reduce the testing time, a memory BIST architecture was proposed in [53] with revised March test algorithms. While most of the previously-described MBIST approaches are focused on testing single port SRAMs, as long as the test algorithms have the features of March algorithms, they are suitable for testing other types of memories with minor modifications. For example, a flash memory BIST architecture was proposed in [49] using a March-like test algorithm. A multiple port SRAM BIST with diagnosis support scheme was introduced in [45] using a modified March algorithm.

In summary, with the exception of the P1500 memory BIST approaches, most of the standalone MBIST architectures focus only on solving the test problems related to a single memory core or a standalone memory chip. They do not account for the specific requirements for integrating the design for test hardware for hundreds of embedded memory cores. They also do not provide any support for test scheduling under power dissipation constraints, which needs a flexible control mechanism for the memory BIST hardware. Although P1500 memory BIST approaches can solve the control problem, a fully-compliant P1500 wrapper and standalone MBIST hardware for all the embedded cores will introduce excessive area overhead and unnecessary performance degradation. To overcome these issues, a new system perspective for memory BIST architectures for complex SOCs is needed. The result turns out to be the distributed MBIST architecture and hardware/software co-testing solutions, as described next.

Distributed MBIST Architecture

To reduce the BIST area and routing overhead as well as the test control complexity associated with complex and heterogeneous SOCs, distributed approaches are necessary. In a distributed memory BIST architecture, each memory core still has a dedicated technology-dependent wrapper. However, depending on the complexity of the SOC, there are only one (or a few) BIST controllers used to direct the test of all the embedded memory cores. Since hardware resource sharing is introduced, to reduce the routing congestion and to facilitate rapid power-constrained testing, the interconnect between the wrappers and the controller(s) must be carefully considered.

Distributed BIST architectures have been advocated for over a decade. Zorian [51] presented a distributed BIST control scheme to test the building blocks of a complex VLSI circuit. Due to the increasing ratio of the memory area in a state-of-the-art SOC, dedicated memory BIST architectures can be used to reduce the cost of memory test. Distributed MBIST architectures can further be divided into: *hardware-centric, software-centric, and hardware/software (HW/SW) co-testing.*

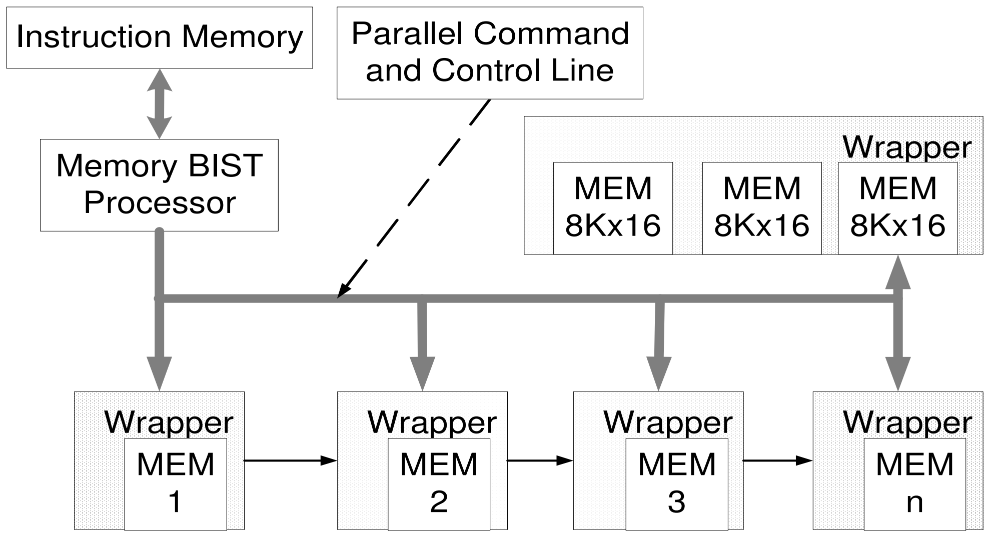


Figure 3.2: A Distributed Memory BIST Approach [7]

1. *Hardware-centric MBIST architecture* : A hardware-centric approach uses dedicated hardware to test all the memory cores in an SOC. It can achieve the near optimum testing time as well as supports flexible test scheduling. However, this approach also introduces large area overhead. A typical *distributed hardware-centric MBIST architecture* was proposed in [7]. As shown in Figure 3.2, each memory (or memory cluster for several identical memories) has a dedicated technology-dependent wrapper. By extracting some technology independent tasks and the test instruction memory to a central controller, which controls all the wrappers, the overall BIST area overhead is reduced. This architecture also integrates several advanced features which have appeared previously in various standalone MBIST approaches. For example, the wrapper can run separate March primitive operations (e.g., $r0$ or $w1$, see Table 2.4 for a detailed list) received from the controller. This implies that the hardware-centric MBIST architecture is programmable and supports multiple March algorithms. Besides, the wrapper design also supports diagnosis by scanning out the faulty addresses and background patterns. However, its main drawback lies in the interconnect between controller and wrappers, which uses one parallel command line to configure all the memory BIST wrappers to run the same test commands (for

example, March primitives in this approach). This implies that for large SOCs, different types of memories (or memories requiring different test algorithms) cannot be tested simultaneously using the same BIST controller, thus increasing testing time as well as test control complexity. Moreover, using parallel interconnects between the controller and the wrappers, the routing congestion may become a potential problem when hundreds of embedded memory cores are present. Furthermore, the testing time for each test session is dominated by the largest memory, which may lead to prohibitively long testing time under power dissipation constraints, as discussed in Section 3.3.

2. *Software-centric MBIST architecture* : A software-centric approach reuses the existing on-chip resources to test all the bus-connected memories. Since SOCs usually contain one or more processing elements, which use on-chip communication architectures to transfer data to/from some of the embedded cores. Reusing these resources for testing the bus-connected memories can lower the area overhead and eliminate the performance penalty caused by the MBIST wrappers. In [32], a methodology for testing SOCs using an on-chip microprocessor was presented. However, this approach uses only software to generate, analyze and apply the test algorithms for the bus-connected memories, which requires a much longer testing time than the existing *hardware-centric* approaches. This is because the hardware architecture can generate March algorithms more efficiently than software. Furthermore, it is obvious that without additional hardware support, the software-centric approaches can only test the bus-connected memories.

3. *Hardware/Software co-testing MBIST architecture* : This architecture takes advantage of both hardware-centric and software-centric approaches. By migrating all the non-time-critical tasks from the MBIST controller to the processor, such as fetching and decoding test instructions, one can reduce the area overhead with a minor testing time penalty. A processor-programmable memory BIST solution was proposed in [38], where a BIST circuit was inserted between the embedded central processing unit (CPU) and the system bus (see Figure

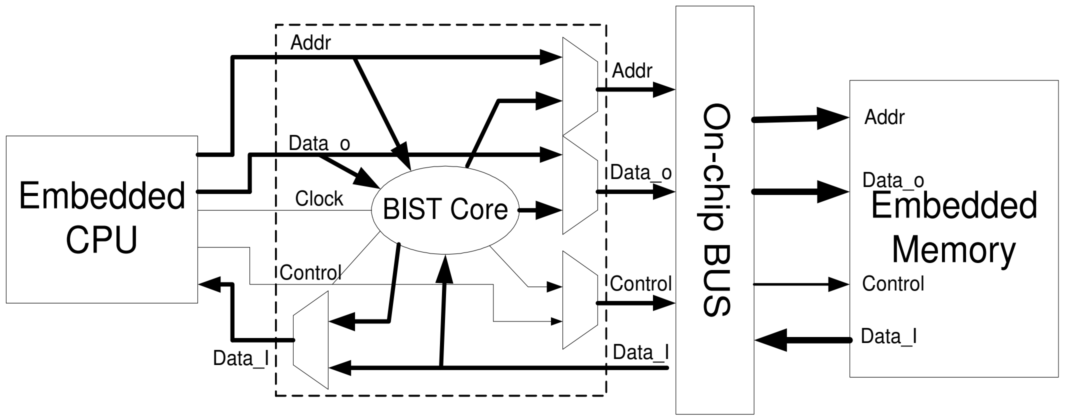


Figure 3.3: Memory BIST for Bus-connected Memories [38]

3.3). Although it reduces the testing time problem associated with the software-centric approach, this solution may affect the overall SOC performance, since the BIST circuitry introduces extra multiplexers between the CPU and the bus, thus increasing the CPU access time to the bus. Moreover, this approach can only test bus-connected memories (BCMs), which is not a complete solution for SOC memory testing.

3.3 Power-Constrained Test Scheduling

In addition to the BIST architectures described in the previous section, test scheduling under power dissipation constraints is becoming an important issue when a large number of memories are embedded in an SOC (see Section 3.1 and Chapter 1 for more details). The test scheduling problem involves both architectural support and test scheduling algorithms for bus-connected memories (BCMs) and non bus-connected memories (NBCMs) .

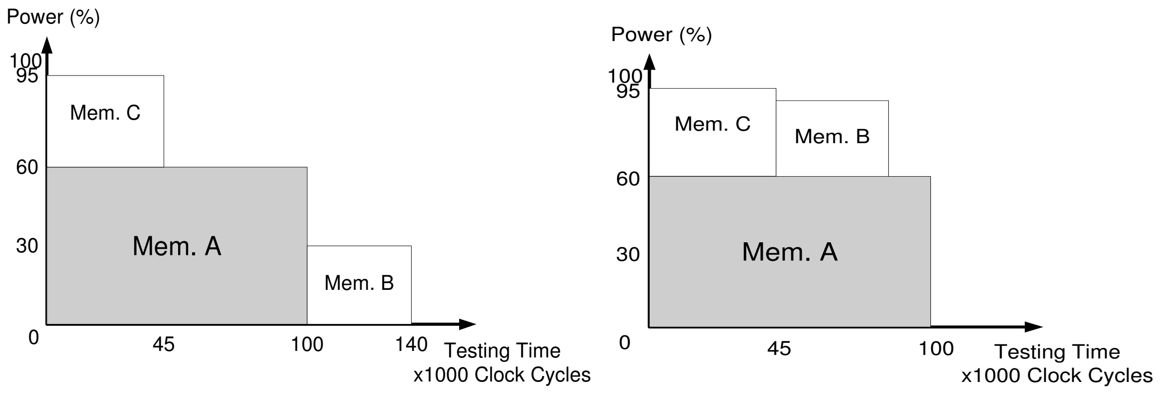
When testing BCMs, because of the resource sharing problem (i.e., different BCMs cannot be accessed simultaneously via the same bus, when using the single-master bus architecture [4]), the power constraints can easily be satisfied since at most one memory is active at a time. For software-centric approaches, such as the one proposed in [32], the testing time will be prohibitively large, which will adversely affect the cost of test. The testing time can be reduced by employing a hardware/software co-testing approach [38] (see Figure 3.3). However, although the time associated with testing each memory is reduced, the serial testing problem is not removed. One potential solution is to wrap some of the non-time-critical bus-connected memories (i.e., a penalty in the memory access time will not influence the overall performance of the SOC) and test them as non bus-connected memories to boost the test concurrency. This is additionally motivated by the fact that not all the embedded memories in an SOC are connected to the system bus. Therefore, hardware-centric memory BIST approaches are necessary for testing these non bus-connected memories. To address power-constrained testing for non bus-connected memories, one effective solution is to limit the number of concurrent memory blocks. The BIST architecture proposed in [7] can be adapted to this solution, however, the testing time of each test session will be dominated by the testing time of the largest memory. This is because this architecture supports only non-partitioned testing [13]. Hence, new flexible BIST architectures need to be investigated, which will guarantee both low area and control complexity, as well as high test concurrency under given power constraints. To achieve this, a control mechanism must be provided to convert non-partitioned testing to partitioned testing with run to completion [13] , as well as to lower both the area overhead and the routing congestion associated with the test control for non bus-connected memories. To illustrate the difference between non-partitioned testing and partitioned testing with run to completion, consider the following example.

Example 3.1 *Let's consider 3 memory blocks A, B and C. Table 3.1 lists the testing time and power dissipation during test for each memory. If non-partitioned testing is used (Figure 3.4(a)), although the test for C is completed before the test for A and the power budget is sufficient to accommodate the test for B, the latter cannot be started until both A and C have completed their tests. When using partitioned testing with run to completion, after the test for C is completed, the test for B can start immediately, as long as the power constraint allows it (see Figure 3.4(b)). Therefore, providing a hardware mechanism that can support partitioned testing with run to completion is essential to achieve a highly concurrent, yet power-constrained, test schedule.*

Most of the existing test scheduling algorithms for SOCs target general IP cores

	Mem A	Mem B	Mem C
Power dissipation (as % of the power constraint)	60%	30%	35%
Testing time(clock cycles)	100,000	40,000	45,000

Table 3.1: Memory Test Parameters for Example 3.1



(a) Non-partitioned Testing

(b) Partitioned Testing With Run to Completion

Figure 3.4: Different Test Schedules for Example 3.1

(i.e., both logic and memory). A representative algorithm based on rectangle packing was proposed in [20]. To exploit the specific characteristics of MBIST architectures, test scheduling algorithms specialized only for memories have started to emerge. Although the algorithm from [44] supports partitioned testing with run to completion [13], it needs to be further improved to deal with both bus-connected and non bus-connected memories when exploiting the particular features of a hardware/software co-testing architecture.

3.4 Special Design Implementation

The detailed design implementation of *all* the modules shown in Figure 3.1 can only be described with a specific architecture and it is beyond the scope of this thesis. What are common, however, to most of the known BIST architectures are the *comparator,* *address generator, and background pattern generator* in the MBIST wrapper.

1. Comparator: The comparator checks the memory output data against the correct background patterns in order to find any mismatch and its implementation is straightforward.

2. Address Generator (AG): The address generator for March-based memory testing has several requirements (see Section 2.3 for details). The most important features of the address generator are that it must cover the entire address space, the internal order of the sequence is irrelevant, however, the down sequence must be in the reverse order of the up sequence. According to these requirements, an automatically synthesized up/down binary counter is sufficient to be the address generator. However, the area of a binary up/down counter is too high for large address spaces [11, 39]. Linear feedback shift registers (LFSR) [5] may overcome this problem, however, since a traditional LFSR does not cover the all 0s pattern, which is necessary for memory testing, it has to undergo some modifications. Furthermore, the LFSR must also be controlled to generate the reversed (down) sequence. A modified LFSR was described in [8, 39] to address these two issues. Another address generator was proposed in [11] to reduce the switching activity on the address lines for power reduction. The activity is minimized when two successive addresses differ in exactly one position. This code sequence is known as Gray-code [16, 43]. However, the area of a Gray-code counter (regardless of the implementation type, i.e., FSM-based or conversion from a binary counter [26]) is much larger than that of an LFSRs[8]. A ripple-carry up/down gray code counter with less area overhead was proposed in [11]. However, this approach has two important limitations. Firstly, the JK flip-flops used in the counter are not always available in vendor's standard cell library [42] which means more gates are needed during synthesis to convert D flip-flops to JK flip-flops. Secondly, for the reflected gray code [16], (see Table 3.2), the down sequence can be generated by flipping the most significant bit (MSB) of the up sequence. Therefore, the up/down control signal for each gray cell in [11] is redundant. Both drawbacks lead to more area and lower speed which is

Up sequence	000	001	011	010	110	111	101	100
Down sequence	100	101	111	110	010	011	001	000

Table 3.2: 3-bit Up and Down Reflected Gray Code Sequence

	Method	**Pros**	**Cons**
Interconnect	Parallel[7]	Low testing time	Routing congestion All wrappers run the same test command
Programmable	March primitive [7]	Any March algorithms	Many instructions
	March element [50]	Fewer instructions	Several March algorithms
Diagnosis	Parallel compression [10]	Low testing time	Large area and many I/O pins
	Serial scan [3]	Low area and I/O pins	Long testing time
HW/SW co-testing	BCM BIST [38]	Fast test BCMs Low area	Can only test BCMs Affect CPU performance
Power	[7]	Support test scheduling	Non-partitioned testing
Scheduling algorithm	Heuristic ordering [44]	Fast	Wrapped memories only
BPG	[47]	None	Complex control / large area
Address generator	Binary	Easy to implement	Large area
	LFSR [8]	Low area	Complex control and design
	Gray counter [11]	Low switching activity	Large area

Table 3.3: Summary of the Existing Solutions for Distributed Memory BIST

crucial for ripple-carry counters.

3. Background Pattern Generator (BPG) : Most embedded memories are word-oriented (i.e., they store more than one bit of data in each address location). In [39], the author listed several background patterns for different fault coverages. Wang and Lee [47] recently presented a hardware implementation for a word-oriented BPG, however, their solution is very complex and has large area overhead. Since there are only $\log_2 N + 1$ states for a BPG, where N is the word-width, we can use a simple FSM to generate all the background patterns very efficiently with much lower area overhead than [47].

3.5 Motivation for New Memory BIST Solutions

The preceding sections have reviewed the relevant previous MBIST approaches and have pointed out several key challenges for power-constrained testing of embedded memories. The features listed in Table 3.3 serve also as the motivation to develop new SOC memory test solutions. In summary, to meet the high quality/low cost objective for SOC memory testing, a new MBIST solution should investigate the following features:

1. *Architecture*: It should be distributed to reduce the area overhead; a serial interconnect scheme between the controller and wrappers can reduce the routing congestion; and MBIST challenges listed in Section 3.1 should be satisfied.

2. *Test scheduling algorithm*: The main challenge is to test both BCMs and NBCMs when using a hardware/software co-testing architecture. By balancing the test access for BCMs between functional resources (on-chip bus) and dedicated DFT hardware (BIST wrapper), one can easily explore various test configurations with different testing times and area overhead under the given power constraints.

3. *Special design implementation*: By analyzing the implementation requirements, new low area background pattern generators and low power ripple-carry gray-code address generators should be investigated.

Although most of the above objectives have been tackled separately by the previous approaches, there are no comprehensive system-level solutions for effective power constrained testing of hundreds of embedded memories (i.e., achieve high test concurrency with low overhead in DFT hardware), that exploit the specific features of SOC architectures. In the following two chapters, two new memory BIST solutions are introduced. Design reuse is also taken into account to ensure that the proposed solutions can easily be integrated as IP blocks into the existing SOC design flows.

Chapter 4

Hardware-centric Memory BIST Architecture

In this chapter, a hardware-centric memory BIST architecture for non-bus-connected embedded memories is introduced. Due to its flexibility (i.e., it is programmable), in addition to reducing routing complexity and easily lowering the power dissipation during test, the presented solution can concurrently support multiple memory test algorithms for heterogeneous memories, it can perform self-diagnosis, as well as embed custom test algorithms required for new memory faults.

4.1 Memory BIST Architecture

The proposed hardware-centric MBIST architecture is shown in Figure 4.1. It consists of a technology-independent memory BIST controller , technology-dependent memory BIST wrappers for heterogeneous memories, and a serial interconnection scheme between the controller and the wrappers. The concept of the serial interconnection is borrowed from IEEE P1500 [23, 24, 30] to reduce the routing congestion. The distinct features of this serail interconnection scheme are outlined next in this chapter. Although the block diagram of this architecture is similar to most of the existing distributed BIST architectures (e.g., [7, 51], by introducing a serial control mechanism and special hardware design techniques, the distinguished feature of the proposed

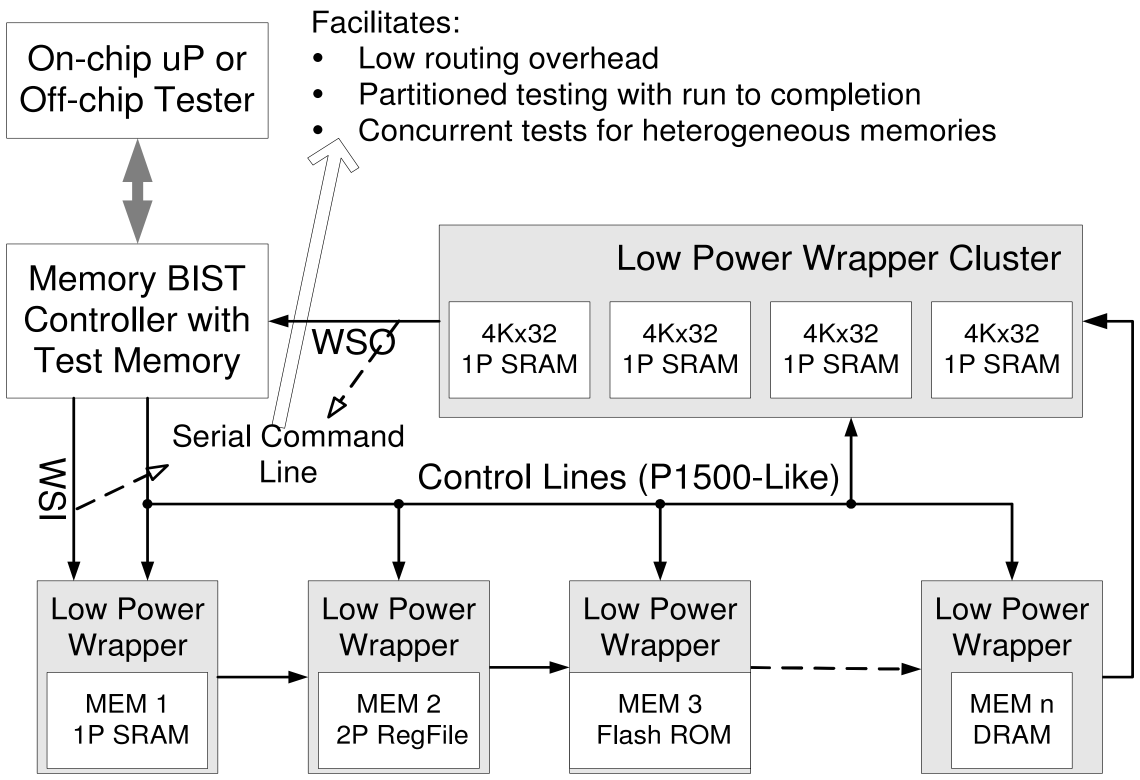


Figure 4.1: New Hardware-centric Memory BIST Architecture

solution is its capability to boost the test concurrency of hundreds of heterogeneous memories under given power constraints, while keeping the hardware overhead low. For example, test instructions sent to the MBIST wrappers via the serial command line, as detailed in the following sections, can be updated at any time while some memories are still under test, which facilitates partitioned testing with run to completion. Since test scheduling for the proposed hardware-centric architecture is a subset of the hardware/software co-testing solution, the scheduling problem will be detailed in Chapter 5. The remaining sections of this chapter provide an in-depth analysis of the hardware implementation of all the building blocks of the proposed hardware-centric BIST solution, followed by experimental results and a summary.

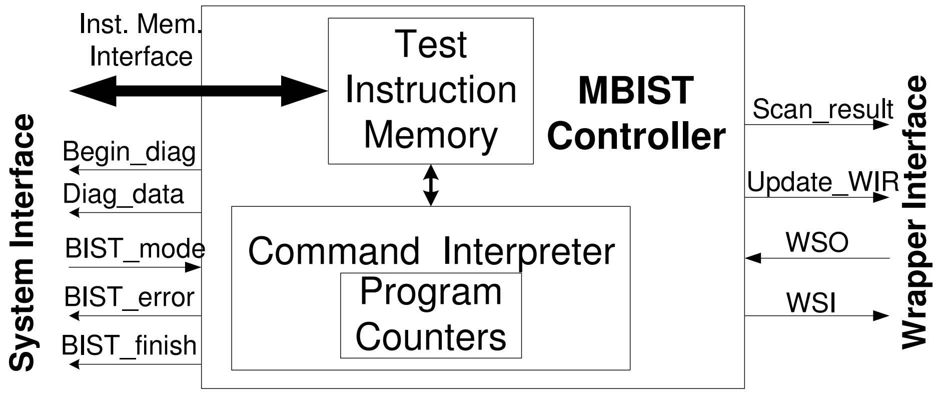


Figure 4.2: Hardware-centric MBIST Controller

4.2 Memory BIST Controller

As shown in Figure 4.2, the MBIST controller is technology independent. It consists of an *instruction memory* that stores the test instructions generated by the test scheduling algorithm (see Chapter 5 for more details) and a *command interpreter*, which executes the test instructions and sends the corresponding commands to each MBIST wrapper. The command interpreter includes a set of *program counters* used to save the location of the March element that is executed for each MBIST wrapper.

The *instruction memory* is a dedicated memory used only for self-test. If it is a ROM no interface for external programming is necessary. If it is RAM then its interface signals are multiplexed with the functional I/O pins. The *instruction memory* is divided into three sub-areas: *algorithm mask, test schedule, and algorithms.* Figure 4.3(a) shows the block diagram of the instruction memory. To reduce the memory size, the test instructions use March elements as the basic commands. This is contrast to using March primitives (or March operations) employed in [7]. In the following each of the sub-areas is explained in detail.

- *Algorithm mask* sub-area assigns each algorithm to one or more memory wrappers. The first word is the mask. There is one bit for each wrapper and '1' means the corresponding wrapper is assigned to this test algorithm. The second word is the address of the algorithm assigned to these memories. If the

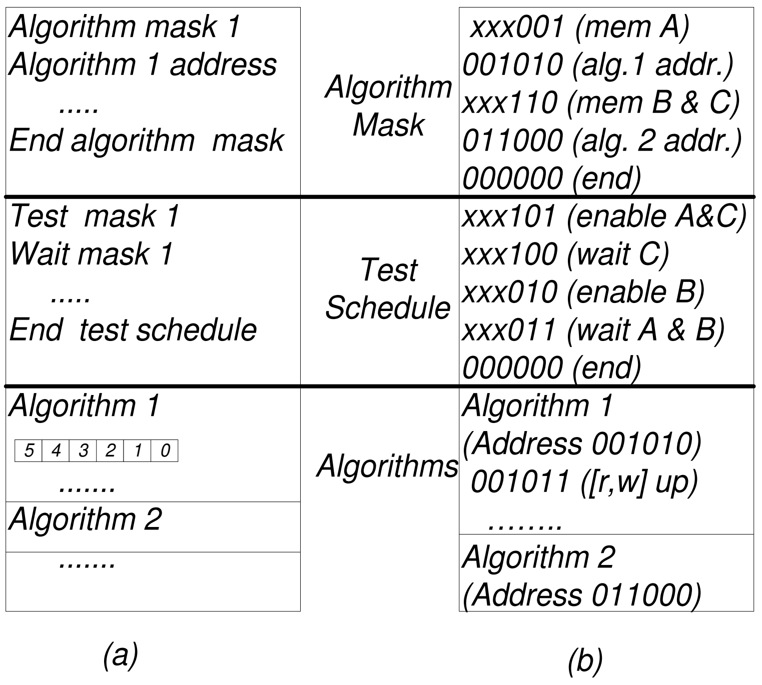


Figure 4.3: Instruction Memory

word width for masks is longer than the memory word width, then it can be stored in several consecutive memory locations. For example, if we have 10 wrappers and the memory word width is 6, then we can use 2 memory locations to keep the mask information.

- *Test schedule* sub-area stores the test schedule for all wrappers connected to this controller. The test scheduling information is generated using the scheduling algorithm (see Chapter 5) and programmed in the instruction memory before the start of the testing process. As shown in Figure 4.3, the first line is the wrapper mask which points out all enabled wrappers running concurrently. This is followed by a wait mask which implies that the current test session must wait until all the wrappers enabled in the wait mask have completed their test, which will trigger the following test session. As indicated for the *algorithm mask* sub-area, if the mask width is longer than the memory word width, then two or more memory locations will be used to store it.

- *Algorithms* sub-area stores all the test algorithms needed for memories directed by the BIST controller. It comprises a list of test instructions, where each test instruction has 6 bits. Bit 0 indicates the test mode, bit 1 defines the up/down address direction and bits 2-5 provide the test command. Each command corresponds to one March element or an alternative (such as reset or next background pattern). Since it was found empirically that 16 test commands (represented on bits 2-5) can cover most of the March algorithms, the word width of this sub-area (and consequently the instruction memory) must be at least 6 bits.

Figure 4.3(b) shows the instruction memory organization for the test scheduling example shown in Figure 3.4(b) (based on Table 3.1 from Section 3.3). Memory A is tested using algorithm 1 and memories B and C are tested by algorithm 2. In the test schedule sub-area, memories A and C are first enabled and using the wait statement it is indicated that only when memory C has completed its test, memory B can be enabled. The testing process will continue until both memories A and B have completed their tests. The two March algorithms are stored in algorithm area.

The *command interpreter* acts as an integer unit in a processor. Its interface consists of a system interface and a wrapper interface. The system interface connects the MBIST controller to the upper level controller (either on-chip microprocessor or off-chip ATE). Through this interface, the upper level controller can activate the testing process and observe the BIST results and diagnosis data from the controller. The wrapper interface links the MBIST controller and MBIST wrappers. Since it includes only a subset of the P1500 signals [23, 24, 30], it has a *P1500-like* serial interconnection. Each MBIST wrapper has a dedicated program counter which points to the currently processed March element (this facilitates multiple March tests to be run simultaneously using a single controller). The main steps for running MBIST are:

1. Program the instruction memory through its interface before the testing process.
2. When the *BIST_mode* signal is asserted (from now onward, unless specified, an asserted signal means it has a logic high level), the controller starts running memory test. The *BIST_mode* signal must be asserted until the controller completes the test and asserts the *BIST_finish* signal.

3. For each test session stored in the instruction memory, the command interpreter will fetch the commands from the algorithm sub-area for all the activated wrappers. Based on their program counter value, the new March elements will be scanned out through the *WSI* port to the wrappers. The *Update_WIR* signal is asserted for one clock cycle after all the commands have been shifted into the corresponding wrappers (see Section 4.3 for further details).

4. After the new commands have been sent, the command interpreter asserts the *Scan_result* signal (see also Section 4.3 for more details) and monitors the response data shifted out from the wrappers through the *WSO* port. If an error occurs, the *BIST_error* signal is asserted to report the error status, and the *Begin_diag* signal is also asserted to indicate that the diagnosis data is ready to be shifted out through *Diag_data* port. If the MBIST is in diagnosis mode, the diagnosis data can be collected, otherwise the self-test is ended immediately after the *BIST_error* signal is asserted.

5. If no error was found or the MBIST is in diagnosis mode, the command interpreter will keep on monitoring the *WSI* port until at least one wrapper has finished running the current March element. Then it will fetch new commands from the instruction memory, de-assert *Scan_result* signal, and repeat the previous two steps until all the test sessions are finished. If an error occurs, the wrapper will continue the current test instruction after the error information is scanned out.

6. After completing the last test command without any error, the *BIST_finish* signal is asserted to denote that all memories under test have passed the test and the upper-level controller can then de-assert the *BIST_mode* signal.

It is essential to note that since the MBIST controller has full controllability/observability of each MBIST wrapper, heterogeneous memories can be tested simultaneously and once a test for a memory has been finished, a new memory test can start immediately. An optional IEEE P1500 [23, 24, 30] interface can also be included in the controller, so the test engineer can control the entire testing process using a standard protocol.

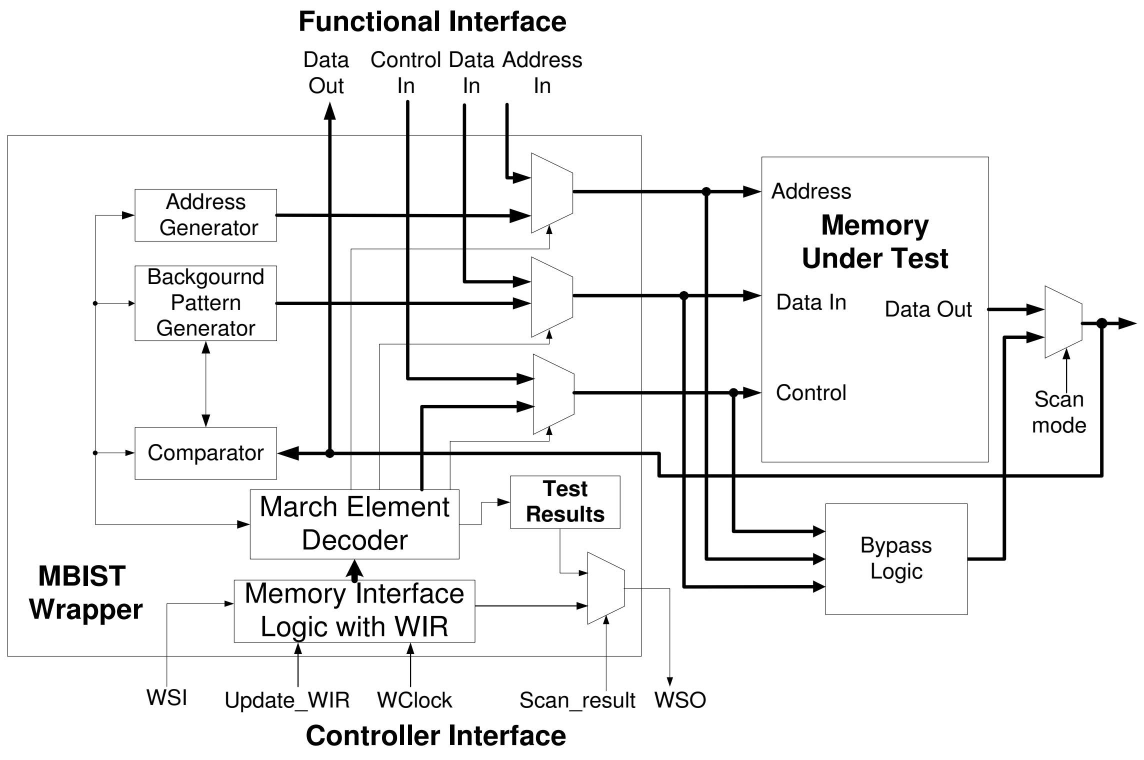


Figure 4.4: Hardware-centric MBIST Wrapper Block Diagram

4.3 Memory BIST Wrapper

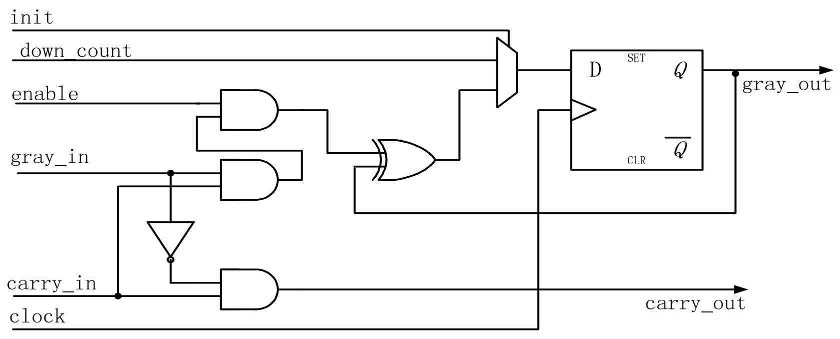
Each embedded memory must be wrapped with a dedicated technology dependent memory BIST wrapper shown in Figure 4.4. To reduce the wire delay and routing congestion, the wrapper must be placed close to the memory core. As stated in Chapter 1, an efficient core wrapper should be able to provide a full controllability and observability for the controller to test the internal core logic as well as to isolate the core and test the interconnect between cores. The IEEE P1500 core wrapper [23, 24, 30] is aimed to provide these capabilities. However, since P1500 wrappers are designed for general core-based SOC testing, a full-compliant P1500 wrapper will induce excessive area and performance degradation. Because of the regularity of memory structure and simple memory I/O interface, a simplified P1500-like memory BIST wrapper is presented to achieve the support of testability and to maintain low area and less performance degradation as well.

On the one hand, to provide the full controllability and observability for the controller to test the memory core, a P1500-like serial scan command line is used. All test commands and test responses are scanned in and out using the *WSI and WSO* ports. The *wrapper instruction register (WIR)* [23, 24, 30] stores the command received from the MBIST controller and using *WClock and Update_WIR* the new command is updated. The *Scan_result* signal controls the scan chain to scan in WIR commands or scan out test results. The *comparator* checks the memory output data for any mismatch. If an error occurs, memory BIST will stop and the controller can then scan out the erroneous address and data pattern stored in the *test results* module for diagnosis purpose. A minor limitation of this scheme is that both wrapper and controller do not store the diagnosis information, rather they just pass it to the upper level controller (see Figure 4.1).

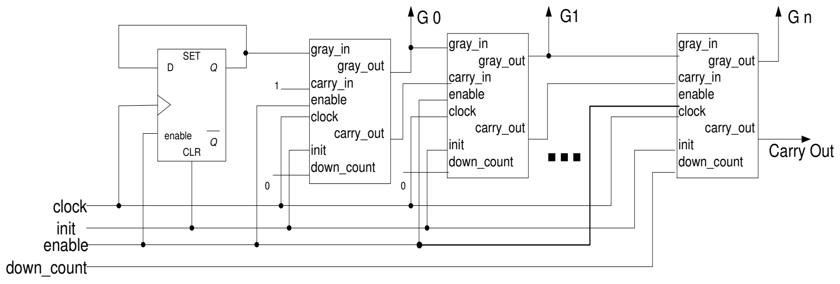
On the other hand, instead of wrapping all I/O ports with wrapper cells [23, 24, 30] in the P1500 wrapper, we only bypass the memory and connect them to other wrappers (or chip I/O pins) and let the test controller to test the interconnect and memory BIST itself. An example is given in Figure 4.4 where all memory inputs are connected to the outputs of the memory through a multiplexer and some bypass logic. In scan mode, this path is enabled to bypass the memory core.

To reduce the manufacturing test complexity, each wrapper has implemented a default March algorithm, which can be activated by a single test command (i.e., no multiple March elements need to be transmitted from the controller). To further reduce wrapper and controller area, identical memory blocks are wrapped as a memory cluster. All the memory blocks in one cluster share the same wrapper (except the comparator sub-block). The wrapper design can support up to 31 memory blocks in one cluster. The test program first enables all the memory blocks in the cluster and, only if an error occurs, after the controller has finished all the test scheduling tasks, it will test each memory block individually to spot the exact faulty memory.

The MBIST wrapper has five main components: *March element decoder, background pattern generator, comparator, address generator, and memory interface logic.* Excepting the comparator, which has a simple straightforward implementation, the



(a) One bit gray counter



(b) N bits gray counter

Figure 4.5: Implementation of the Reflected Gray Code Counter

other four components are detailed in the following.

March Element Decoder

The March element decoder is based on an FSM that decodes the test commands received from controller and generates the control signals for memory read/write and address and background pattern generation. The implementation is determined by the memory specifications (e.g., type and size) and the requirements for test. To keep its size under control, the decoder implements only a few March elements, which is sufficient to run most of the known March algorithms.

Address Generator

The address generator is a simple ripple-carry gray-code up/down counter that is tailored for March-based memory test only. As introduced in Chapter 3, to build an up/down gray-code counter as a memory address generator, we only need to have a normal up sequence gray counter. Using the reflected gray code, the down sequence can be obtained by reversing the MSB bit of the up sequence. The detailed implementation of this address generator is shown in Figure 4.5. Figure 4.5(a) shows the one bit gray cell modified for March-based memory testing. Figure 4.5(b) illustrates a n-bit ripple-carry gray-code counter, which includes one dummy D flip-flop used to control the least significant bit. The *carry_in* signal is connected to the *carry_out* signal of the previous gray cell. When the last bit of *carry_out* is asserted, it means that the counter has finished counting; the *gray_in* signal is connected to the *gray_out* signal of the previous cell which is also the output address bit; the counter will update its state when the *enable* signal is asserted; the *down_count* signal is connected to '0' for all the gray cells except the MSB bit to force the up sequence when it is de-asserted or the down sequence when it is asserted; the *init* signal initializes the counter to the starting value, which is all '0's when *down_count* is de-asserted and '1000...0' sequence when *down_count* is asserted. The approach helps to achieve low switching activity on the address lines (and hence low power in the address logic decoder and memory cells) using less area overhead than other approaches. However, to implement the gray-code counter as an address generator, one restriction is that the memory address space must be a power by 2, which is a practical assumption.

Background Pattern Generator

Most of the embedded memories are word-oriented, i.e., they contain more than one bit per word and need to be tested multiple times with different background patterns. Since there are only $\log_2 N + 1$ states for a background pattern generator (BPG) , where N is the word width, we can use a simple decoder with predefined values to generate all the background patterns. An example pseudo-code to generate 8-bit word background patterns (shown in Table 2.6 in Chapter 2) is listed below:

8-bit Word Background Pattern Generator

INPUT: INIT, NEXT_PATTERN
OUTPUT: PATTERN, REV_PATTERN

```
1. if (INIT) {
2.     PATTERN = 8'b00000000; REV_PATTERN = 8'b11111111;
3.     STATE = 0;
4. } else if (NEXT_PATTERN) {
5.     STATE = STATE + 1;
6. }else {
7.     case (STATE) {
8.        0: PATTERN = 8'b00000000; REV_PATTERN = 8'b11111111;
9.        1: PATTERN = 8'b01010101; REV_PATTERN = 8'b10101010;
10.       2: PATTERN = 8'b00110011; REV_PATTERN = 8'b11001100;
11.       3: PATTERN = 8'b00001111; REV_PATTERN = 8'b11110000;
12.    }
13. }
14. done
```

In this example, an 8-bit word background pattern has $\log_2 8 + 1 = 4$ states, which means we only need a 2-bit register (variable $STATE$ in the pseudo-code shown above) to store the 4 states. A valid $NEXT\_PATTERN$ signal increments the $STATE$ register (line 4-6). Based on the value of the $STATE$ register, a simple 4-1 multiplexer (case statement from line 7 to 13) selects the output pattern $PATTERN$ and the reversed pattern $REV\_PATTERN$ from the pre-coded pattern values.

Memory Interface Logic

To reduce the routing overhead for the proposed distributed architecture, a serial interface between the controller and the wrappers is used. IEEE P1500 [23, 24, 30] is

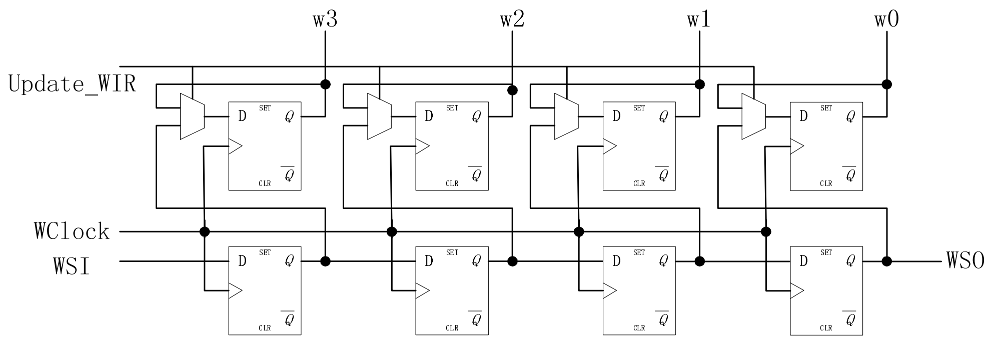


Figure 4.6: Wrapper Instruction Register (WIR)

a standard core test wrapper which has a serial interface, however both the area and performance overhead for a fully P1500-compliant wrapper is large. Since the interconnect between controller and wrappers is not shared with any other logic cores, a very simple serial *IEEE P1500-like* circuit that only implements the WIR and the serial interface (in addition to an architecture-specific *Scan_result* signal), is introduced to reduce the area and performance overhead associated with the wrapper boundary register (WBRs) used to wrap all the ports in standard IEEE P1500 approach.

Figure 4.6 shows the internal structure of a 4-bit WIR. It has two register arrays, one is connected to the serial command line to scan-in the command, and the other is used to store the command when *Update_WIR* signal is asserted. Based on this structure, the controller can scan-in different commands for different memories under test. In other words, the controller has full controllability/observability for each memory block. This approach needs a slightly longer time to send the test commands (when compared to the parallel command line architecture). This additional scan-in time is determined by total number of WIR bits of all the wrappers. When compared to the total memory testing time, this minor overhead is neglectable (detailed experimental results can be found in Section 4.4).

Having introduced all the components in an MBIST wrapper, a summary of the main steps, used to run a test command in the wrapper, are given.

1. The wrapper will gain the control of the memory core after it receives a valid test command from the controller through the serial command line. When

processing the current test command stored in WIR, the wrapper is locked by the *March element decoder* and therefore no new test commands can be updated during this period. The only way to interrupt the test is to send a *reset* command to this wrapper or reset the entire system.

2. Based on the received test commands, the *March element decoder* will generate the appropriate signals to control the *address generator, background pattern generator, and comparator* used to run the March test algorithm. The *test results* module stores the results to scan them out to the controller. It will be either a success flag if no error occurs, or an error packet (erroneous address, data, and an error flag) if an error occurs.

3. When an error occurs, the wrapper stops and waits for the controller to scan out the error packet. If the test is in the diagnosis mode, the test will resume after the error packet is scanned out. Otherwise, the memory testing process is ended with a fail.

4. After finishing the current test command, the test results are shifted out. The WIR is then unlocked and ready for a new test command.

4.4 Experimental Results

A set of experiments have been performed using high-density embedded synchronous SRAMs [41] compiled for 0.18μ CMOS technology [37]. The wrapper is configured to implement 9 March elements: *(w), (r), (rw), (rwr), (rww), (rwww), (rwrwr), (rwrwrw), (wwrww)*, which are the building blocks for most of the known March test algorithms [39, 47]. The last March element *(wwrww)* is used to run the word-oriented March C- test (*March-CW* proposed in [46]).

The efficiency of the proposed BPG is illustrated in Table 4.1. The proposed BPG has only one third of the area required by the BPG described in [47]. Table 4.2 compares the proposed gray code up/down address generator with a directly synthesized binary up/down counter (for the same performance constraints). In addition to saving 30% of the area required by an address generator, the power dissipation during

Width	**Proposed** (μm^2)	**[47]** (μm^2)	**Difference**
8	382	1,273	233.25%
16	793	2,395	202.02%
32	1,102	4,622	319.42%

Table 4.1: Background Pattern Generator Area

Width	**Proposed** (μm^2)	**Binary** (μm^2)	**Difference**
8	1,350	1,853	37.26%
12	2,106	2,772	31.62%
16	2,781	3635	30.71%
20	3,448	4,614	31.62%
24	4,098	5,899	33.82%

Table 4.2: Address Generator Area

test is also decreased. For an N-bit binary counter the total number of transitions is $2^{N+1} - N - 2$ [11], while for an N-bit gray counter the total number of transitions is only 2^N. Therefore, the switching activity for the address bus is reduced by approximately 50% and hence both peak and average power will be decreased, thus facilitating higher test concurrency under power constraints.

A comparison of the area overhead between the proposed memory BIST wrapper and the IEEE P1500 full-compliant wrapper [23, 24, 30] together with memory BIST circuit is shown in Table 4.3. The BIST area overhead (BAO) is very low and when the embedded memory size increases the wrapper area overhead will further decrease (for both single memories and memory clusters). However, with the P1500 wrapper, the BAO will be over 50% larger than the proposed single memory BIST wrapper. For clustered memories, the BAO of the P1500 wrapper is several times larger than that of the proposed approach. This is because all memories in a cluster share one MBIST wrapper, however, each of them must have their own P1500 wrapper to support internal and external testability [23, 24, 30]. Table 4.4 shows the comparisons of the overall BIST area overhead with three different approaches: (1) the proposed memory BIST wrapper, (2) the IEEE P1500 full-compliant wrapper [23, 24, 30] together with memory BIST circuit, and (3) the proposed memory BIST

Mem.	**Area(μm^2)**	**Wrap.(μm^2) MBIST**	**BAO**	**Wrap.(μm^2) MBIST +P1500**	**BAO**
512x16	93,346	11,119	11.91%	16,998	18.21%
1kx32	263,685	14,079	5.34%	24,044	9.12%
4kx32	884,509	14,681	1.66%	24,882	2.81%
4-Block Cluster					
4x512x16	373,384	11,794	3.16%	37,410	10.02%
4x1kx32	1,054,740	15,038	1.43%	57,398	5.44%
4x4kx32	3,538,036	15,604	0.44%	58,907	1.66%

Table 4.3: MBIST (With and Without P1500) Wrapper Area Overhead

wrapper which use binary address generator and BPG from [47]. Note, the BAO for the MBIST controller includes also a 64x8 test program memory. The BAOs vary for different configurations. Again, the BAO of P1500 compliant wrapper is much larger than that of the proposed MBIST wrapper. Although the BAO using the proposed approach with binary address generator and BPG from [47] is only slightly larger than the proposed solution. It should be noted that we only have maximum 5 memory cores in this example. The use of the proposed gray-code address generator and BPG will contribute to both overall BAO and power savings when hundreds of memory core are embedded in an SOC. From these two figures, it is also clear to see that the BAO for configurations with memory clusters is always less than that with same memories but without memory clusters.

Figure 4.7 shows the area overhead of the address generator, background pattern generator, MBIST wrapper, and MBIST controller for different configurations. Note that the width of the BPG (Figure 4.7(b)) is varied by a power of 2. It is easy to see that the BAOs of all these blocks increase linearly when varying the parameter on the horizontal axis (i.e., address space, word width and wrapper number). This implies that the percentage of the additional design for test-related area will decrease as the number of memory cores increases in an SOC. In terms of gate size, it is clear to see that the MBIST controller consumes the largest portion of the total BIST

Configuration	**1kx32+4kx32**	**1kx32x4+4kx32**	**4x1kx32+4kx32**
Memory(μm^2)	1,148,194	1,939,249	1,939,249
BIST(μm^2)	64,960	114,260	65,919
BAO	5.66%	5.89%	3.40%
BIST+P1500(μm^2)	85,125	164,320	118,491
BAO	7.41%	8.47%	6.11%
BIST+BPG in[47] +Binary AG(μm^2)	69,117	120,628	70,076
BAO	6.02%	6.21%	3.61%

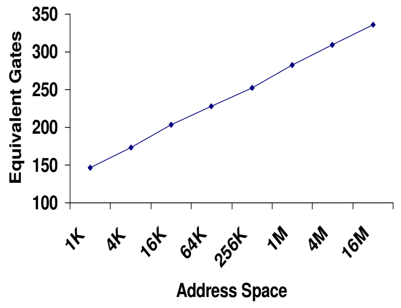
Table 4.4: Comparisons of Total BAO for Different Approaches

4kx32 + 2 1kx32 Using March-CW [46]			
	Optimal (ns)	**Proposed (ns)**	**[7] (ns)**
Test Time	1,474,560	1,480,540	>1,843,200
Penalty	0%	0.4%	>25%
Configuration			
Mem Size	4kx32	1kx32	1kx32
Power(%)	66%	30%	30%

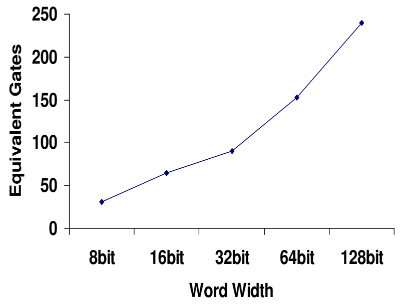
Table 4.5: Testing Time Comparison

area. Therefore, in addition to decreasing the potential performance penalty caused by placing MBIST wrappers around bus connected memories, a key challenge is to reduce the area of the controller, which is the motivation for the development of the hardware/software co-testing MBIST architecture described in the next chapter.

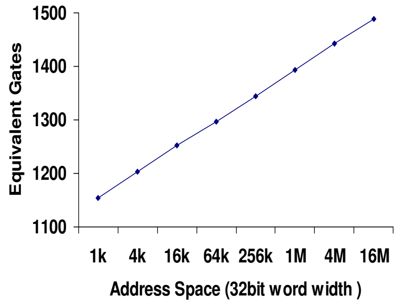
Detailed results on test scheduling are given in the following chapter. At this time, to illustrate the difference between non-partitioned testing supported by [7] and partitioned testing with run to completion [13] supported by the proposed architecture, three NBCMs are used: one 4kx32 and two separate 1kx32 memories. All the memories use the March-CW test algorithm proposed in [46]. Table 4.5 shows the testing time comparison, where the optimal solution assumes separate test control for all the wrappers, which will obviously lead to maximum test concurrency at the expense of high area and performance penalty. On the one hand, without considering the time used to send commands and receive test results, the testing time for the approach



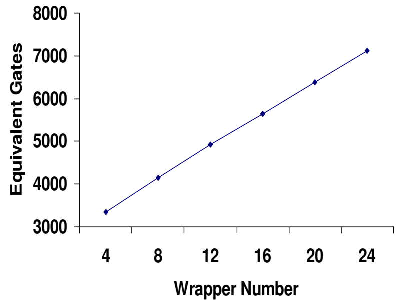
(a) Address Generator BAO



(b) BPG BAO



(c) Wrapper BAO



(d) Controller BAO

Figure 4.7: BIST Area Overhead for Each Component

from [7] is at least 25% higher than the optimal testing time. On the other hand, due to the explicit support for partitioned testing and the low extra time required to scan in commands and scan out test results, the proposed solution introduces insignificant penalty in testing time when compared to the optimal solution.

4.5 Summary

The cost of test when using memory BIST is determined primarily by the *testing time, BIST area overhead, and routing congestion.* It was shown through experimental results that the proposed distributed architecture can reduce the BIST area overhead;

scan-based serial interconnection reduces the routing congestion; the support for partitioned testing with run to completion can reduce the testing time by more efficient scheduling; special low power design of MBIST wrappers can further reduce the area overhead, as well as, due to reduced activity, increase test concurrency under power constraints, which will ultimately lower the testing time. Due to its programmability, it can support multiple March algorithms, and heterogeneous memories can be tested at the same time using different or custom March algorithms [2]. In addition, the diagnosis feature aids the failure analysis process by providing the location of the faulty cells, which can help adjust the manufacturing process and/or search for improved test algorithms for the technology at hand.

Chapter 5

HW/SW Co-testing Memory BIST Architecture

Chapter 4 has presented a hardware-centric solution that can concurrently support multiple memory test algorithms for heterogeneous memories with relatively low area and routing overhead. It can also perform self-diagnosis, as well as support partitioned testing with run to completion [13]. Since this solution does not assume that the SOC includes processing elements connected to embedded memories through on-chip busses, it is independent of the SOC structure and it is equally applicable to any large digital chip with heterogeneous embedded memories. However, in practice, an SOC contains one or more processing elements (e.g., microprocessors), which use on-chip system busses to communicate with some embedded memory cores. Reusing these existing on-chip resources to direct the self-testing process can further lower the BIST area overhead, as well as eliminate the potential performance penalty caused by wrapping the bus-connected memories. In this chapter, a new hardware/software co-testing memory BIST architecture is presented. By exploiting the existing on-chip resources to test the embedded memories, a design space exploration methodology, based on a test scheduling engine, is described. The proposed solution is capable to reduce the area of the BIST controller, eliminate the wrapper area overhead and potential performance degradation for bus-connected memories, while preserving all the advantages of the hardware-centric approach presented in the previous chapter.

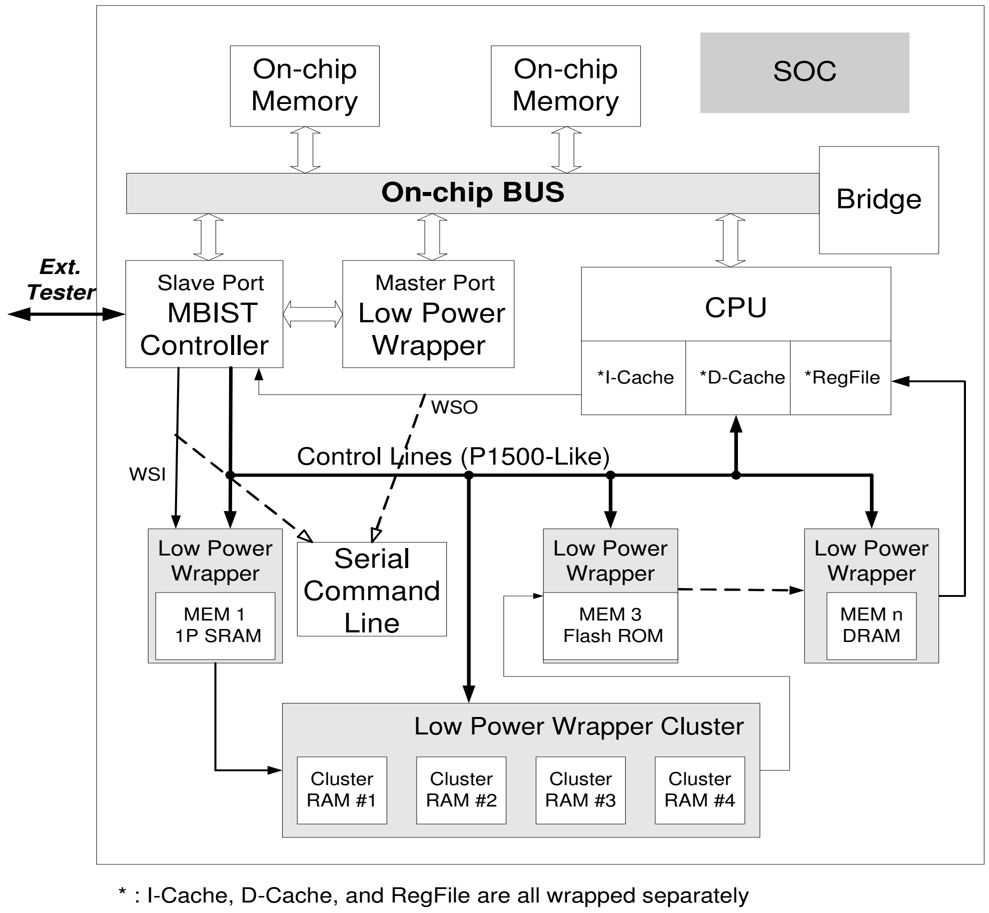


Figure 5.1: New HW/SW Co-testing Memory BIST Architecture

5.1 Memory BIST Architecture

This section details the hardware structure of the new MBIST architecture for *hardware/software co-testing* , which can be defined as the process of consciously partitioning the test access, test application and test control functions for embedded memories between dedicated DFT hardware and the available on-chip functional resources.

The proposed memory BIST architecture (see Figure 5.1) consists of two components: the programmable MBIST controller and the MBIST wrappers . There are two types of wrappers: wrappers connected to the system bus to test bus-connected memories (BCMs) and individual wrappers for each non-bus-connected memory (NBCM) or a cluster of identical NBCMs. The NBCM wrapper is identical to the one described in section 4.3. However, the BCM wrapper can be shared by all the memories on the same bus and it is a new component specific to the architecture described in this chapter. The MBIST controller is also redesigned to support hardware/software co-testing and to test both BCMs and NBCMs. To test BCMs, unlike the approach reported in [38], the proposed solution uses the standard bus interface to exchange data between the CPU and the MBIST controller, and hence it can affect only the bus performance. Furthermore, if the critical path of an SOC is not on the system bus (which is a realistic case in practice), the presented solution will not influence the SOC performance. In addition, by using a standard bus interface, the proposed MBIST module can be reused as an soft IP core. The test mechanism for NBCMs is the same as for the hardware-centric MBIST architecture. It also supports multiple March algorithms, can perform self-diagnosis, as well as facilitates partitioned testing with run to completion[13] for power-constrained test scheduling. In addition, by running most of the non-time-consuming tasks, such as fetch and decode of test commands, in software using an embedded microprocessor, the BIST area overhead of the MBIST controller can be reduced.

A comparison of the most important features supported by the hardware/software co-testing and hardware-centric MBIST architectures is shown in Table 5.1. The main advantage of the hardware-centric approach lies in its independency of the SOC structure (after the initialization and activation it can run the memory testing process without any assistance). Since all the memories are wrapped, the testing time may be lower under power constraints. However, the introduction of the instruction memory and wrappers for every memory core leads to large area overhead and performance penalty. By reusing the embedded microprocessor for test instruction decoding and a shared MBIST wrapper for all the memories connected to the same bus, the hardware/software co-testing MBIST architecture can eliminate the dedicated instruction memory and the wrappers for BCMs. This approach may require slightly longer testing time if all the memories attached to system busses are unwrapped, however it

	Hardware-centric	**HW/SW co-testing**
Architecture	Independent Low testing time High area and performance overhead	Interact with embedded processor Low/medium testing time Low area and performance overhead
Controller	Diagnosis support Run multiple March algorithms concurrently Support partitioned testing with run to completion	
	Complex with instruction memory Control all NBCM wrappers	Simple without instruction memory Control BCM and NBCM wrappers
NBCM wrapper	Diagnosis support Support multiple March algorithms Test memory or memory cluster concurrently Serial interconnection between controller and wrapper	
BCM wrapper	N/A	Diagnosis support Support multiple March algorithms Can test all BCMs serially Parallel interconnection
Design implementation	Area efficient background pattern generator Low area ripple-carry gray code up/down address generator Reusable IP with standard bus interface	
Scheduling algorithm	Partitioned with run to completion	Partitioned with run to completion Wrap BCMs to reduce testing time

Table 5.1: Comparison of Hardware-centric and HW/SW Co-testing MBISTs

provides great savings in BIST area and may improve the system performance. A detailed description of the implementation of the hardware/software co-testing MBIST architecture is given next.

Novel Programmable MBIST Controller

The block diagram of the programmable MBIST controller is shown in Figure 5.2. It consists of three interface logic modules for on-chip bus, BCMs and NBCMs respectively. The *bus interface module* provides a standard bus slave interface to connect the controller to the system bus and exchange data with the microprocessor for both BCMs and NBCMs. To test BCMs, the *BCM interface module* passes all the test commands received from the processor through bus interface to the BCM wrapper , and sends back the test results (pass/fail and diagnosis information) to the processor.

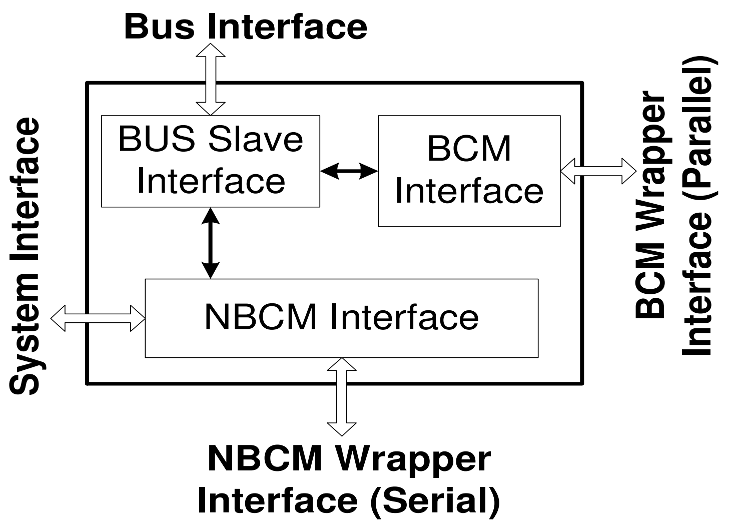


Figure 5.2: Memory BIST Controller

The connection between BCM wrapper and MBIST controller is parallel to reduce the control complexity since they are both connected to the bus and can be physically placed very close to each other. A detailed explanation of the BCM interface signals and test procedures for BCMs can be found in Section 5.1. The *NBCM interface module* is designed to test all the NBCMs in the SOC. This module has two parts: a system interface to connect the controller with the off-chip ATE and a NBCM wrapper interface to hook-up all the NBCM wrappers using a serial scan chain. These parts are identical to the ones presented for the hardware-centric approach presented in Chapter 4. To test NBCMs using an on-chip processor, the *NBCM interface module* first receives parallel test commands from the processor, performs parallel/serial conversion and sends serial commands to all the NBCM wrappers and then performs serial/parallel conversion for the results received from NBCM wrappers before passing them to the processor. However, before the embedded processor can run memory tests, we need to have a fault-free memory to store the test program and test results. This memory can be either on-chip or off-chip. If the fault free memory is off-chip then the processor can directly fetch test commands from the off-chip memories without any ATE support. If the fault free memory is on-chip (e.g., processor's cache) then it must be pre-tested using an ATE via the system interface.

In terms of diagnosis, the processor can directly get the diagnosis information of BCMs using the parallel bus interface and then send it off-chip for failure analysis. The diagnosis information of NBCMs is directly shifted out using the system interface. By running most of the non-time consuming tasks, such as fetching and decoding the test commands and analyzing the response data, in the processing unit using software, the BIST area overhead of the MBIST controller is decreased when compared to [7] and the hardware-centric MBIST approach introduced in Chapter 4. In addition, by reusing cache memories or other pre-tested memories to store the test instructions, the dedicated instruction memory (see Figure 4.3) is also removed, thus leading to further savings in area.

Memory BIST Wrappers

Both BCM wrappers, illustrated in Figure 5.3, and NBCM wrappers (see Figure 4.4 in Chapter 4 for details) have a common hardware implementation that is capable to run March-based memory testing algorithms [39]. These include: address generator, background pattern generator, comparator, and a March element decoder to interpret commands received from the MBIST controller and generate appropriate control sequence to perform memory testing. Detailed information on these four common components can be found in Section 4.3. The difference between BCM and NBCM wrappers lies in the interconnect interface between the controller, wrappers, and memories under test.

The BCM wrapper uses a bus master interface to test a part (or all) of the memories connected to the bus (depends on the test scheduling constraints described in Section 5.3). The interconnect interface between wrapper and the bus can be parallel without any concerns for routing congestion. To speed up the data exchange through parallel interconnects, the BCM wrapper can also be placed (in the physical floorplan) close to the MBIST controller. By using the common test access resource (functional bus), one BCM wrapper can test all BCMs serially. Without considering power dissipation constraints, this may affect the overall testing time. However, by carefully

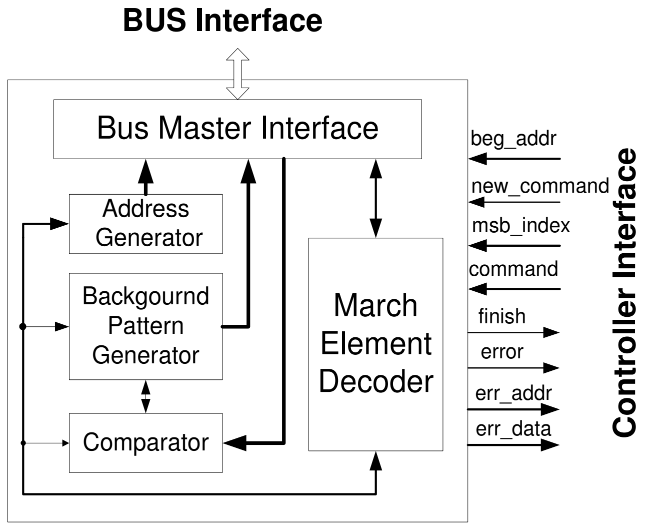


Figure 5.3: Bus-connected Memory BIST Wrapper

scheduling the test sessions under power constraints (detailed in Section 5.3), one can reduce simultaneously both the testing time and the area overhead by unwrapping as many BCMs as possible. The signals of the interface between the BCM wrapper and the controller are shown in Figure 5.3. All of them are directly mapped to some bus-accessible addresses that can be read or written by the processor through the slave port in the MBIST controller. The address generator in the BCM wrapper should cover the address space of the largest memory connected to the bus. For each BCM, the begin address and the size of the memory is first sent to the BCM wrapper to guide the address generator to generate the proper addresses. Since BCMs usually cover the entire address space (i.e., the address space is a power of 2), the ripple-carry gray-code up/down counter proposed in hardware-centric architecture can be used. The index of the MSB is used to indicate the memory size. For example, MSB index 9 means that the memory has 512 words. These requirements also lead to slight differences between the test procedures for the BCM wrapper and NBCM wrapper introduced in Chapter 4. When using a parallel interface between BCM wrapper and the controller, the test procedure for BCMs is introduced below.

1. The processor sends memory begin address, memory size, and test command to the BCM wrapper through *beg_addr, msb_index, and command* signals.

2. The controller then asserts *new_command* signal to activate the BCM wrapper in order to execute the test command (instruction).

3. If the test command is finished without an error, the *finish* signal is asserted. If an error occurs, the *error* signal is asserted and the *err_addr and err_data* are set to the erroneous address and data pattern for diagnosis.

4. After the current test command is finished, the processor will send a new command through the slave port of the controller to the BCM wrapper until all the scheduled instructions are completed.

Despite the differences in interconnect between BCM and NBCM wrappers, both of them have the distinctive benefits described in Section 4.3 (e.g., can run multiple March algorithms or and support self-diagnosis).

5.2 Software Implementation

The memory BIST architecture described in the previous section can facilitate hardware/software co-testing for a given test schedule generated using the algorithm presented in the next section. This section details the software implementation necessary to handle the test control flow. The test software is written in C (rather than in assembly language) to make it easily portable to any SOC environment. First, the test commands are generated using a power-constrained test scheduling algorithm (see Section 5.3) and loaded to a instruction memory (I-cache or other memories where the test instructions are stored), which was pre-tested using ATE. The test software then fetches commands from the instruction memory and sends data (reads responses) to (from) the MBIST controller. Since the time-consuming tasks (e.g., on-chip generation of March tests) are conducted by the dedicated BIST hardware, using a high level specification language for test control will not affect the overall testing time. In the following the pseudo-code is given.

Hardware/Software Memory Testing

```
1. Test processor and bus;
2. Test cache memories and register files using ATE;
3. Load test program to I-Cache, test data to D-Cache;
6. Do {
7.     Read commands from instruction memory;
8.     Send commands to MBIST controller;
9.     Wait for response;
10.     if (error) {
11.        if (diagnosis enable) {
12.           if (BCM error) read diagnosis information
13.           else wait for NBCM diagnosis information to scan out;
14.        } else break;
15.     }
16. }Until all the scheduled test sessions are completed;
```

The software implementation supports two modes of operation: manufacturing test mode and diagnosis mode. In the manufacturing test mode, only one self-test command is needed for each wrapper, which will automatically run the embedded default March algorithm and set the fail/pass bits after completing the testing process. When in the diagnosis mode, the CPU can send any March element supported by the wrapper to run different March algorithms and read back the results (address, background pattern) for further diagnosis purposes. In addition to aiding fault diagnosis, this is a very suitable mechanism to find the best March algorithm for a new technology or to increase the test quality when the current default March algorithm cannot achieve the targeted defect coverage.

5.3 Test Scheduling

In this section, the test scheduling engine, which is also used to explore different hardware/software co-testing configurations is introduced. Since the SOC bus architecture will affect the way how the BCMs are accessed during test, it will also affect the test schedule and ultimately the testing time. If only one bus master can access the bus, all the BCMs must to be tested sequentially. If the SOC has several large BCMs, this SOC bus architecture will adversely affect the testing time. To address this drawback, one can use alternative bus architectures, such as multi-layered AMBA bus [4], to which several bus masters can be attached and non-shared slave components can be accessed concurrently. By adding one or more BCM wrappers with bus master interface, this architecture supports more flexible test scheduling, thus reducing the testing time. However, if no multi-layered SOC bus architectures are present for the native mode of execution, another solution would be to wrap some BCMs, which have a critical impact on testing time, with dedicated wrappers and test them as NBCMs. Due to our practical validation setup, for the test scheduling algorithm presented in this section and in our experimental results, we have considered the second option (i.e., to boost test concurrency we transform some BCMs to NBCMs), however, note, the same trade-off exploration engine can be used if multi-layered SOC bus architectures are employed.

Problem Formulation

Recently a test scheduling formulation and algorithm have been reported in [44] only for NBCMs. The test scheduling problem for the proposed architecture is a generalization of the one presented in [44], since in our case we have both BCMs and NBCMs. For NBCMs the constraints are the same as in [44]. However, to account for the specific test features of BCMs, each memory connected to a bus will have different resource conflict relationship and values for testing time: one when it is wrapped with dedicated wrapper (i.e., when it is transformed to a NBCM to increase test concurrency) and one when it time-shares the wrapper connected using a master port to the SOC bus architecture. The scheduling algorithm automatically chooses which BCM will be transformed to NBCM, thus exploring various test configurations with different testing time and area overhead. The parameters used in the algorithm are listed below:

P_{max}	power constraint during test;
n_m	total number of BCMs and NBCMs;
n_b	total number of busses;

for $i \in 1, 2, ..., n_m$

p_i	maximum power dissipation for memory i;
lw_i	testing time for memory i when wrapped;
luw_i	testing time for memory i when unwrapped;
s_i	scheduled test start time for memory i;
l_i	scheduled testing time for memory i;

Formulation:

Objective: Minimize $T_{max} = MAX(s_i + l_i)$

Subject to: $\sum p_i \leq P_{max}$ where i are the memories currently under test;

Test Scheduling Algorithm

Since test scheduling has been proven to be an NP-complete problem [13], in this section we propose a greedy heuristic to deal with complex SOCs comprising hundreds of memories. The aim of the algorithm is to reduce the testing time under power dissipation constraints while lowering also the area overhead, via unwrapping all the BCMs without affect the testing time. The intuition beyond using the test scheduling algorithm as a design space exploration engine that decides how many BCM are wrapped can be explained as follows. If all the BCM memories are unwrapped and share a single BCM wrapper then the test concurrency is limited by both resource sharing (i.e., the same functional bus) and power constraints. If dedicated NBCM wrappers are used for BCMs then test concurrency will also be limited by power constraints. Therefore, the proposed exploration engine searches for the highest number of embedded memories connected to the system bus that can share the same BCM wrapper without limiting the test concurrency imposed by power constraints.

The Algorithm $Mem\_Schedule$ takes the test parameters of each memory (p_i, lw_i, luw_i) and the power constraint (P_{max}) as inputs, and it outputs the test schedule $T_{schedule}$ and the test method for each memory core M_{test} (i.e., wrapped or unwrapped). Note, this algorithm also increases the number of unwrapped BCMs, without affecting the testing time. The test of each memory is represented as a rectangle whose width is the testing time and whose height is power dissipation. For BCMs two rectangles with different testing time are necessary (one for each test method), while for NBCMs one rectangle representation is sufficient. If two BCMs on the same single-master bus are unwrapped, then they cannot be tested at the same time due to bus contention, and we call these two memories incompatible. The proposed algorithm is based on the rectangle packing algorithm $TAM\_schedule\_optimizer$ proposed in [20].

The algorithm starts by initializing each memory wrapper test method M^i_{test}: all memories are treated as wrapped except those explicitly specified as unwrapped. Then the testing time T^i_{time} of each memory is computed based on its test method (line 2). The currently available power constraint P_{avl} is initialized to P_{max} and the number of unscheduled memories is initialized to the total number of embedded memories (line 3). As long as there is an unscheduled memory, the algorithm first finds a compatible memory m_i with maximum testing time max_{tat}, which does not exceed the power dissipation constraint (line 6). If such a memory m_i exists, it will be scheduled and the available power dissipation constraint will be updated (line 8, 9). If no compatible memories meet the power constraint, we will record the idle power dissipation P_{idle}, update the available power dissipation $P_{avl} = 0$ (line 11) and branch to the end of the schedule of the memory with the minimum end time min_{end}. Afterward we update the new schedule information, including the new schedule begin time, the number of unscheduled memories $N_{unscheduled}$ and available power dissipation P_{avl} (line 12-15). If there is still some idle test time, we will look for the already scheduled memories and check whether we can unwrap them without affecting the testing time (line 16, 17). The algorithm exits when all the memories have finished their schedule. Although the proposed test scheduling algorithm supports partitioned testing with run to completion [13], it can easily be adapted to non-partitioned testing used by other MBIST architectures [7] (in line 12 the schedule of m_j is finished with max_{end}).

Memory Schedule Algorithm *Mem_Schedule*

INPUT: M, P_{max}
OUTPUT: $T_{schedule}$, M_{test}

```
1. Initialize $M^i_{test}$;
2. Compute $T^i_{time}$;
3. Initialize $P_{avl} = P_{max}$; $N_{unscheduled} = N_{memories}$
4. while ($N_{unscheduled} != 0$) {
5.    if ($P_{avl} > 0$) {
6.       find compatible $m_i$ with $max_{tat}$ and $P_i < P_{avl}$;
7.       if (found) {
8.          schedule $m_i$;
9.          $P_{avl} -= P_{m_i}$;
10.      } else {
11.         $P_{idle} = P_{avl}$; $P_{avl=0}$;
.        }
.    } else {
12.      Finish schedule of $m_j$ with $min_{end}$;
13.      Update schedule begin time;
14.      $N_{unscheduled}$--;
15.      $P_{avl} += P_{m_j}$; $P_{avl} += P_{idle}$
16.      if (idle time exists) {
17.         unWrap scheduled mems if not incompatible;
.        }
.     }
.  }
18. done
```

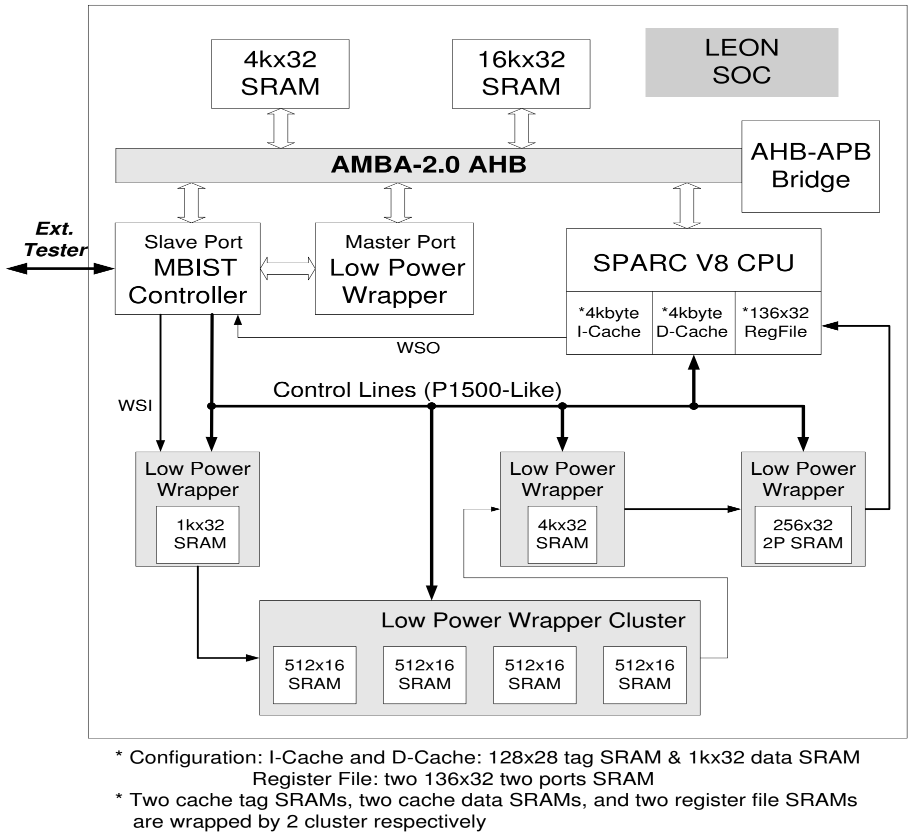

4kx32 SRAM
16kx32 SRAM
LEON SOC
AMBA-2.0 AHB
AHB-APB Bridge
Ext. Tester
Slave Port MBIST Controller
Master Port Low Power Wrapper
SPARC V8 CPU
*4kbyte I-Cache
*4kbyte D-Cache
*136x32 RegFile
WSO
WSI
Control Lines (P1500-Like)
Low Power Wrapper 1kx32 SRAM
Low Power Wrapper 4kx32 SRAM
Low Power Wrapper 256x32 2P SRAM
Low Power Wrapper Cluster
512x16 SRAM
512x16 SRAM
512x16 SRAM
512x16 SRAM
* Configuration: I-Cache and D-Cache: 128x28 tag SRAM & 1kx32 data SRAM
Register File: two 136x32 two ports SRAM
* Two cache tag SRAMs, two cache data SRAMs, and two register file SRAMs are wrapped by 2 cluster respectively

Figure 5.4: Example LEON SOC Platform Configuration

5.4 Experimental Results

To prove the correctness and the effectiveness of the proposed MBIST architecture, that facilitates hardware/software co-testing, we have implemented and fabricated an SOC platform based on LEON [17]. LEON is an open source core for embedded applications and it has an IEEE-1754 (SPARC V8) [35] compatible integer unit with 8 register windows, 4KB I-Cache, 4KB D-Cache, and a full implementation of the AMBA-2.0 bus [4]. A detailed illustration of this SOC platform is shown in Figure 5.4.

To estimate and compare different test parameters (e.g., BIST area overhead, testing time, performance penalty, power dissipation) of the proposed solution, a set of

	BCM	**NBCM**	**Total**
Memory (μm^2)	4,100,930	2,698,855	6,799,785
Wrapper(μm^2)	14,449	109,248	123,697
Controller (μm^2)	9,684 / BAO = 0.14%		
BIST Area Overhead (%)	0.35	4.04	1.96

Table 5.2: MBIST Area Overhead for LEON SOC

	Software Centric	**Programmable BIST Core**	**Hardware Centric**
Testing Time	7.0	1.4	1.0
BIST Area	0	Low	High
Performance	0	Affect bus speed	Affect mem. speed
Flexibility	Any March	Most March	Fixed

Table 5.3: Different Approaches for Testing BCMs.

experiments have been performed using high-density embedded SRAMs compiled for 0.18μ CMOS technology. In our LEON-based SOC platform, the NBCMs are a cluster of four 512x16 SRAMs, one 1Kx32 SRAM and one 4Kx32 SRAM, while the BCMs are one 4Kx32 SRAM and one 16Kx32 SRAM. The wrappers have been configured to implement 9 March elements: *(w), (r), (rw), (rwr), (rww), (rwww), (rwrwr), (rwrwrw), (wwrww)*, which are the building blocks for most of the known March test algorithms [39, 47]. The last March element *(wwrww)* is used to run the word-oriented March C- test (*March-CW* proposed in [46]). All wrappers also implement March-CW algorithm as the default memory test algorithm. Table 5.2 shows the BIST area overhead for the LEON SOC. By adding a very low area MBIST controller (0.14% in this case), the CPU can control self testing for most of the embedded memories (note, cache memories and the register file are pre-tested using ATE as outlined in section 5.2). The MBIST area overhead (control + wrappers) of this LEON SOC is less than 2%. Timing analysis indicates that the critical path is in the CPU core, which means that our approach does not introduce performance degradation. However, the NBCM wrappers do affect the memory access speed.

Table 5.3 shows the comparison of three different approaches (software-centric,

Memory Size	**Word Width**	**Power (mw/Hz)**
128	8	0.035
128	16	0.054
128	32	0.091
512	16	0.047
512	32	0.097
1024	16	0.050
1024	32	0.132
2048	16	0.121
2048	32	0.219
4096	16	0.151
4096	32	0.265
8192	16	0.198
8192	32	0.351
16384	16	0.284
16384	32	0.500

Table 5.4: Memory Configuration and Power Dissipation [41]

hardware-centric, and programmable BIST core attached to the bus to be used with hardware/software co-testing) for bus connected memories using the LEON platform [17] and applying the March-CW algorithm [46]. As shown in the table, the proposed solution combines the benefits of the other two approaches. It requires low area overhead, it may affect only the bus performance, and it maintains the flexibility of the software-centric approach, while it reduces the testing time significantly, bringing it close to the best possible testing time given by the hardware-centric approach.

Using the proposed MBIST architecture and the greedy heuristic for test scheduling described in Section 5.3, Table 5.5 gives a comparison between partitioned testing with run to completion and non-partitioned test scheduling algorithms under power dissipation constraints. The memory cores included in this experiment are of different sizes with the number of address lines ranging from 7 to 16 and the word size ranging from 8 to 32. The power dissipation for these cores ranges from 3.5 mW to 50 mW for 100 MHz clock frequency. The parameters (size, word width, and power dissipation) of memory cores used in this test scheduling experiment are listed in Table 5.4.

Memory Cores	**10**	**30**	**50**	**70**	**100**	**150**	**200**	**250**	**300**
Maximum Power Dissipation = 250mW									
Partitioned	156672	986112	1299456	1612800	2082816	3382272	4165632	5465088	7944192
Non-Partitioned	156672	1004544	1317888	1631232	2101248	3400704	4184064	5497344	7976448
Difference(%)	0.00	1.83	1.40	1.13	0.88	0.54	0.44	0.59	0.40
Unwr.(part.)	1	0	0	0	2	3	5	6	7
Unwr.(non-part.)	1	2	4	6	9	14	19	23	25
Maximum Power Dissipation = 500mW									
Partitioned	73728	589824	663552	737280	893952	1419264	1704960	2276352	3428352
Non-Partitioned	82944	764928	903168	1032192	1248768	1672704	2022912	2916864	3700224
Difference(%)	11.11	22.89	26.53	28.57	28.41	15.15	15.72	21.96	7.35
Unwr.(part.)	1	1	1	0	2	0	0	4	1
Unwr.(non-part.)	1	3	5	7	10	15	20	25	19
Maximum Power Dissipation = 1000mW									
Partitioned	73728	589824	589824	589824	589824	829440	1105920	1382400	1972224
Non-Partitioned	73728	700416	829440	958464	1124352	1437696	1732608	2045952	2654208
Difference(%)	0.00	15.79	28.89	38.46	47.54	42.31	36.17	32.43	25.69
Unwr.(part.)	1	0	1	0	0	0	0	2	0
Unwr.(non-part.)	1	5	7	9	12	16	21	25	26

Table 5.5: Testing Time (cc) and Wrapped Memories for Different Test Schedules

All the memories are compiled using the Virage Logic memory compiler [41] and the maximum power dissipation was obtained from the data sheet. On the one hand, it can be seen that the maximum testing time of non-partitioned testing is always greater than (or at least equal to) the partitioned testing with run to completion. The difference between the two testing times varies based on the maximum power constraint and memory configurations. For example, the more we relax the constraints, i.e., higher test concurrency can be achieved, the difference in testing time increases, unless the maximum concurrency has been achieved by both algorithms. One the other hand, because there is more idle time in non-partitioned testing, more BCMs can be unwrapped and tested serially using the on-chip bus. This means that for non-partitioned test scheduling we may increase the testing time at the benefit of lower BIST area overhead. It should be noted that the proposed greedy heuristic is very fast (e.g., for 300 memories it takes less than 1 second).

	min.	**max.**	**diff.**
Memories=100 Power=250mW			
Testing Time(part.)	2082816	2709504	30.09%
BCM as NBCM(part.)	18/20	0/20	18
Testing Time(non-part.)	2101248	2912256	38.60%
BCM as NBCM(non-part.)	11/20	0/20	11
Memories=300 Power=250mW			
Testing Time(part.)	7944192	10672128	34.34%
BCM as NBCM(part.)	53/60	0/60	53
Testing Time(non-part.)	7976448	10994688	37.84%
BCM as NBCM(non-part.)	35/60	0/60	35

Table 5.6: Testing Time vs. Wrapper Numbers.

Finally, the trade-off between BIST area overhead and testing time is shown in Table 5.6. Item *"BCM as NBCM"* means the BCM has a dedicated wrapper and it will be tested as NBCM. To decrease BIST area overhead, i.e., all BCMs are unwrapped, the testing time will be increased. Similarly, to reduce the testing time, most of the BCMs have to be wrapped thus increasing the BIST area overhead. Using the test scheduling engine, as a trade-off exploration tool, is beneficial especially when the testing time for the entire SOC is dominated by the scan testing time of embedded logic cores. In this case, we can explore different hardware/software co-testing configurations until we match the logic cores' testing time with the lowest DFT area required by dedicated MBIST wrappers.

5.5 Summary

This chapter has introduced a comprehensive embedded memory test solution based on a new hardware/software co-testing memory BIST architecture. By running the non-time-critical tasks in an embedded processor, the area of the BIST controller is reduced. In addition, a new test scheduling algorithm is used as a design space exploration engine and it can simultaneously schedule the memories under test and decide how many bus connected memories need to be wrapped. Exhaustive experimental results have demonstrated the effectiveness of the proposed solution.

Chapter 6

Conclusion

As more and more memory cores are embedded in state-of-the-art SOCs, embedded memory testing has emerged as a key issue in the VLSI design, implementation and fabrication flow. Low power, low area overhead, low routing congestion, low testing time and reduced performance overhead are a few key issues that need to be tackled. To achieve this, two new distributed memory BIST approaches have been introduced in this thesis. The first approach is called *hardware-centric memory BIST architecture* . It can concurrently support multiple test algorithms for heterogeneous memories with relatively low area and routing overhead. It can perform self-diagnosis , as well as support partitioned test with run to completion [13] scheduling , which will ultimately lead to a reduction in testing time. This hardware-centric approach is totally independent of the SOC structure and therefore it is suitable for any large digital circuits with embedded memories, regardless whether processing elements and bus structures exist or not on the chip.

Since state-of-the-art SOCs contain one or more processing elements, which use on-chip system busses to communicate with the memory cores, reusing these existing on-chip resources for testing the bus-connected memories can lower the BIST area overhead as well as eliminate the performance degradation caused by the BIST wrappers. This motivates the second approach, a *new hardware/software co-testing memory BIST architecture* . By reusing the existing on-chip resources (bus-based communication architectures and software stored in the cache and executed on the embedded CPU), this architecture further reduces the area overhead of the BIST controller, eliminates the wrapper area overhead and performance degradation for bus-connected memories, in addition to all the advantages of the first approach. A greedy test scheduling algorithm developed specifically for embedded SOC memory testing can further increase the flexibility of the hardware/software co-testing approach. By partitioning the number of wrappers added to bus-connected memories, using the proposed scheduling algorithm, less silicon area can be consumed to meet certain testing time and power dissipation constraints. The experimental results prove the benefits of the novel features of both approaches.

The research findings presented in this thesis have been validated using two manufactured circuits that include SRAMs. Although these architectures are applicable to heterogeneous memories, the existing restriction is that test algorithms must be March-based. For some types of memories, such as large embedded DRAMs, March-based test algorithms are not sufficient to detect certain physical defects. Therefore, a future work can improve the proposed architectures to support both March-based and non March-based test algorithms for embedded DRAM-specific memory faults, such as SNPSFs (see Chapter 2). Moreover, since the yield of large embedded memory cores is very low [53], built-in memory repair is currently emerging as a necessary technology to increase the overall SOC yield. Since the architectures proposed in this thesis do not support self-repair implicitly, an additional future improvement will be to extend the proposed architectures with explicitly support for self-repair.

Appendix A

Silicon Implementation

To empirically validate the proposed architectures using real silicon instead of simulation, the proposed BIST architectures have been included in two digital chips and submitted for fabrication through CMC [9].

Figure A.1 shows the micro photo of the fabricated chip that implements the *hardware-centric MBIST architecture*. The chip is implemented in TSMC [37] 0.18μ CMOS technology with high density embedded synchronous RAMs (SRAMs) provided by Virage Logic [41]. The total silicon space of this chip is 2000x2000 micron^2 and is granted by CMC [9]. It embeds two 1kx32 SRAMs, four 256x32 SRAMs as a memory cluster and one 256x32 SRAM to store the test instructions. The fabricated chip was tested using the IMS/ATS1 test and validation system [14] and it was proven that it can correctly perform the designated memory BIST tasks.

The second approach, *hardware/software co-testing MBIST architecture* was implemented on a 2000x2500 micron^2 die. It comprises the LEON-2 (version 1.0.9) [17] SOC platform. Given the the limited space allocation, it includes only 4k bytes I-cache and D-cache memory, 8 register windows, and two 256x32 bus-connected memories. This chip is still under testing, however the back-annotated simulations have shown that it can perform the designated BIST functions. Figure A.2 illustrates the layout view of this chip.

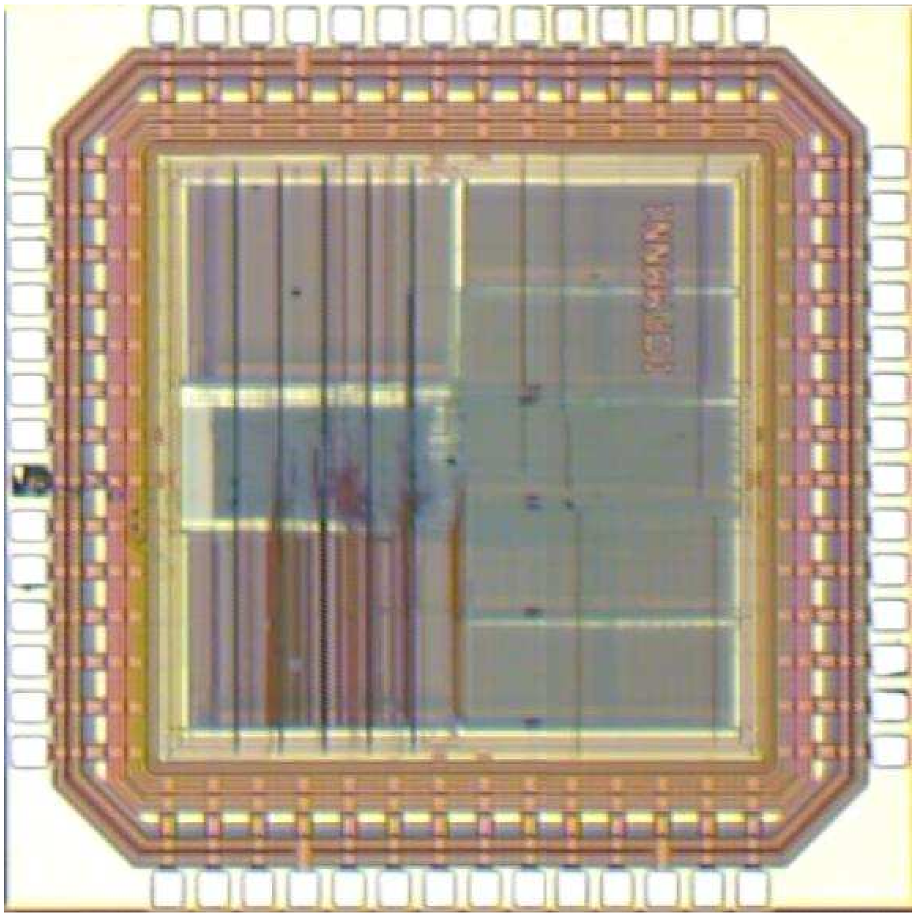


Figure A.1: Microphoto of Hardware-centric MBIST Architecture

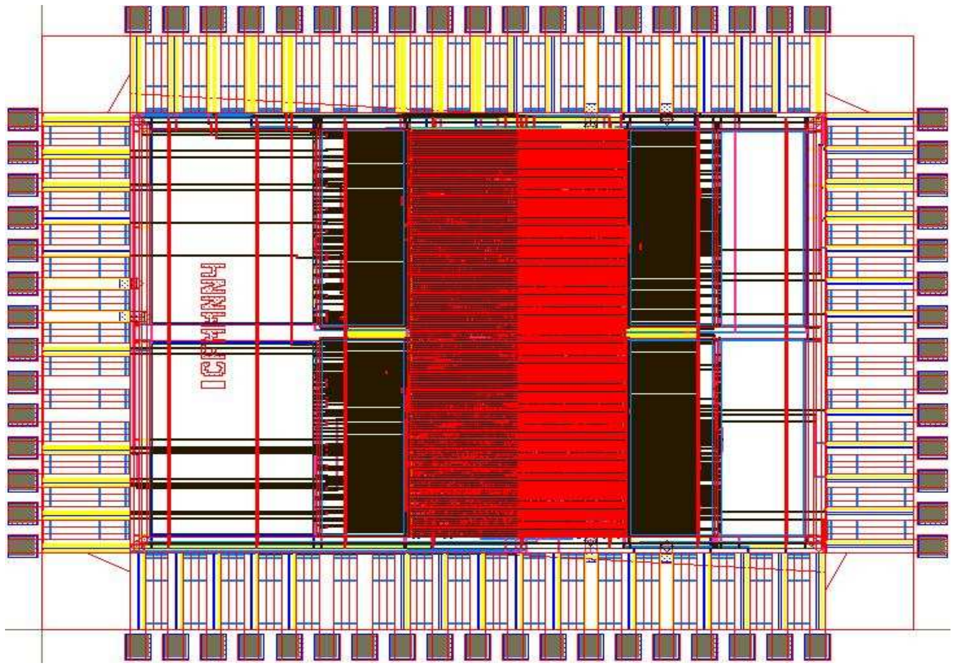


Figure A.2: Layout View of Hardware/Software Co-testing MBIST Architecture

Bibliography

[1] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1995.

[2] S. M. Al-Harbi and S. K. Gupta. An Efficient Methodology for Generating Optimal and Uniform March Tests. In *Proc. IEEE VLSI Test Symposium*, pages 231–237, 2001.

[3] D. Appello, F. Corno, M. Giovinetto, M. Rebaudengo, and M. S. Reorda. A P1500 Compliant BIST-Based Approach to Embedded RAM Diagnosis. In *Proc. IEEE Asian Test Symposium*, pages 97–102, 2001.

[4] ARM Inc. AMB Web Site. http://www.arm.com.

[5] P. H. Bardell, W. H. McAnney, and J. Savir. *Built-In Test for VLSI: Pseudo-random Techniques*. John Wiley & Sons, Inc., New York, 1987.

[6] T. J. Bergfeld, D. Niggemeyer, and E. M. Rudnick. Diagnostic Testing of Embedded Memories Using BIST . In *Proc. of the Design, Automation and Test in Europe Conference*, pages 305–309, 2000.

[7] M. L. Bodoni, A. Benso, S. Chiusano, S. D. Carlo, G. D. Natale, and P. Prinetto. An Effective Distributed BIST Architecture for RAMS. In *Proc. IEEE European Test Workshop*, pages 119–124, 2000.

[8] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing*. Kluwer Academic Publishers, 2000.

[9] Canadian Microelectronics Corporation . CMC Web Site. http://www.cmc.ca.

[10] J. T. Chen, J. Rajski, J. Khare, O. Kebichi, and W. Maly. Enabling Embedded Memory Diagnosis via Test Response Compression. In *Proc. IEEE VLSI Test Symposium*, pages 292–298, 2001.

[11] H. Cheung and S. K. Gupta. A BIST Methodology for Comprehensive Testing of RAM with Reduced Heat Dissipation. In *Proc. IEEE International Test Conference*, pages 386–395, 1996.

[12] R. M. Chou, K. K. Saluja, and V. D. Agrawal. Scheduling Tests for VLSI Systems Under Power Constraints. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(2):175–184, June 1997.

[13] G. L. Craig, C. R. Kime, and K. K. Saluja. Test Scheduling and Control for VLSI Built-In Self-Test. *IEEE Transactions on Computers*, 37(9):1099–1109, September 1988.

[14] Credence Systems Corporation. Credence Web Site. http://www.credence.com.

[15] J. Dreibelbis, J. Barth, H. Kalter, and R. Kho. Processor-Based Built-In Self-Test for Embedded DRAM. *Solid-State Circuits*, 33(11):1731–1740, November 1998.

[16] F. Gray. Pulse Code. U.S. Patent Application 94 111 237.7, Mar. 1953.

[17] Gaisler Research. LEON Web Site. http://www.gaisler.com.

[18] P. T. Gonciari, B. M. Al-Hashimi, and N. Nicolici. Test Data Compression: the System Integrator's Perspective. In *Proc. of the Design, Automation and Test in Europe Conference*, pages 726–731, 2003.

[19] International SEMATECH. *The International Technology Roadmap for Semiconductors (ITRS): 2001 Edition.* http://public.itrs.net/Files/2001ITRS/Home.htm, 2001.

[20] V. Iyengar, K. Chakrabarty, and E. J. Marrinissen. On Using Rectangle Packing for SOC Wrapper/TAM Co-Optimization. In *Proc. IEEE VLSI Test Symposium*, pages 253–258, 2002.

[21] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-On-A-Chip Designs*. Kluwer Academic Publishers, third edition, 2002.

[22] S. Koranne, C. Wouters, T. Waayers, S. Kumar, R. Beurze, and G. S. Visweswaran. A P1500 Compliant Programmable BistShell for Embedded Memories. In *Proc. IEEE International Workshop on Memory Technology, Design and Testing*, pages 21–27, 2001.

[23] E. J. Marinissen, R. Kapur, M. Lousberg, T. McLaurin, M. Ricchetti, and Y. Zorian. On IEEE P1500's Standard for Embedded Core Test. *Journal of Electronic Testing*, 18(4):365–383, August 2002.

[24] E. J. Marinissen, Y. Zorian, R. Kapur, T. Taylor, and L. Whetsel. Towards a Standard for Embedded Core Test: An Example. In *Proc. IEEE International Test Conference*, pages 616–627, 1999.

[25] P. Mazumder and K. Chakraborty. *Testing and Testable Design of High-Density Random-Access Memories*. Kluwer Academic Publishers, 1996.

[26] H. Mehta, R. M. Owens, and M. J. Irwin. Some Issues in Gray Code Addressing. In *Proc. IEEE VLSI Great Lakes Symposium*, pages 178–181, 1996.

[27] W. M. Needham. Nanometer Technology Challenges for Test and Test Equipment. *Computer*, 32(11):52–57, November 1999.

[28] N. Nicolici and B. M. Al-Hashimi. *Power-Constrained Testing of VLSI Circuits*. Kluwer Academic Publishers, Frontiers in Electronic Testing (FRET) Series, 2003.

[29] Open Verilog and VHDL International . Accellera Web Site. http://www.accellera.org.

[30] P1500 SECT Task Forces. IEEE P1500 Web Site. http://grouper.ieee.org/groups/1500.

[31] M. Powell, S. H. Yang, B. Falsafi, K. Roy, and N. Vijaykumar. Reducing Leakage in a High-Performance Deep-Submicron Instruction Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):77–89, February 2001.

[32] R. Rajsuman. Testing a System-on-a-Chip with Embedded Microprocessor. In *Proc. IEEE International Test Conference*, pages 499–508, 1999.

[33] R. Rajsuman. Design and Test of Large Embedded Memories: An Overview. *IEEE Design and Test of Computers*, 18(3):16–27, May-June 2001.

[34] J. Saxena, K. M. Butler, V. B. Jayaram, S. Kundu, N. V. Arvind, P. Sreeprakash, and M. Hachinger. A Case Study of IR-Drop in Structured At-Speed Testing. In *Proc. IEEE International Test Conference*, pages 1098–1104, 2003.

[35] Sparc International Inc. Sparc Web Site. http://www.sparc.org/standards/V8.pdf.

[36] Synopsys Inc. Synopsys Web Site. http://www.synopsys.com.

[37] Taiwan Semiconductor Manufacturing Corporation. TSMC Web Site. http://www.tsmc.com.

[38] C. H. Tsai and C. W. Wu. Processor-Programmable Memory BIST for BUS-Connected Embedded Memories. In *Proc. Asia and South Pacific, Deisng Automation Conference*, pages 325–330, 2001.

[39] A. J. van de Goor. *Testing Semiconductor Memories: Theory and Practice.* A.J. van de Goor, 1998.

[40] V. A. Vardanian and Y. Zorian. A March-Based Fault Location Algorithm for Static Random Access Memories. In *Proc. IEEE International Workshop on Memory Technology, Design and Testing*, pages 62–67, 2002.

[41] Virage Logic. Virage Logic Web Site. http://www.viragelogic.com.

[42] Virtual Silicon Technology, Inc. Virtual Silicon Web Site. http://www.virtual-silicon.com.

[43] J. F. Wakerly. *Digital Design Principles and Practiecs.* Prentice Hall, third edition, 2001.

[44] C. W. Wang, J. R. Huang, Y. F. Lin, K. L. Cheng, C. T. Huang, C. W. Wu, and Y. L. Lin. Test Scheduling of BISTed Memory Cores for SOC. In *Proc. IEEE Asian Test Symposium*, pages 356–361, 2002.

[45] C. W. Wang, R. S. Tzeng, C. F. Wu, C. T. Huang, C. W. Wu, S. Y. Huang, S. H. Lin, and H. P. Wang. A Built-In Self-Test and Self-Diagnosis Scheme for Heterogeneous SRAM Clusters . In *Proc. IEEE Asian Test Symposium*, pages 103–108, 2001.

[46] C. W. Wang, C. F. Wu, J. F. Li, C. W. Wu, T. Teng, K. Chiu, and H. P. Lin. A Built-In Self-Test and Self-Diagnosis Scheme for Embedded SRAM. In *Proc. IEEE Asian Test Symposium*, pages 45–50, 2000.

[47] W. L. Wang, K. J. Lee, and J. F. Wang. An On-Chip March Pattern Generator for Testing Embedded Memory Cores. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(5):730–735, October 2001.

[48] V. N. Yarmolik, Y. V. Klimets, A. J. van de Goor, and S. N. Demidenko. RAM Diagnostic Tests. In *Proc. IEEE International Workshop on Memory Technology, Design and Testing*, pages 100–102, 1996.

[49] J. C. Yeh, C. F. Wu, K. L. Cheng, Y. F. Chou, C. T. Huang, and C. W. Wu. Flash Memory Built-In Self-Test Using March-Like Algorithms. In *Proc. IEEE International Workshop on Electronic Design, Test and Applications*, pages 137–141, 2002.

[50] K. Zarrineh and S. J. Upadhyaya. On Programmable Memory Built-In Self-Test Architectures. In *Proc. of the Design, Automation and Test in Europe Conference*, pages 708–713, 1999.

[51] Y. Zorian. A Distributed BIST Control Scheme for Complex VLSI Devices. In *Proc. IEEE VLSI Test Symposium*, pages 4–9, 1993.

[52] Y. Zorian, E. J. Marinissen, and S. Dey. Testing Embedded-Core Based System Chips. In *Proc. IEEE International Test Conference*, pages 130–143, 1998.

[53] Y. Zorian and S. Shoukourian. Embedded-Memory Test and Repair: Infrastructure IP for SoC Yield. *IEEE Design and Test of Computers*, 20(3):58–66, May-June 2003.

Index