Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Generic and Orthogonal March Element based Memory BIST Engine

Halil Kukner

CE-MS-2011-25

Abstract

The International Technology Roadmap for Semiconductors (ITRS) forecasts that embedded memories will dominate the System-on-Chip (SoC) area by about 94% in no more than five years. In addition, the manufacturing process in the nano-era introduces new and complex failure mechanisms. The detection of such failures requires at-speed testing. Moreover, the complexity of electronic systems has reached a level where the accessibility of embedded memories using external test equipment is getting harder and costly, if not infeasible. Thus, high quality memory testing is crucial to improve the overall SoC quality.

Built-in Self-Test (BIST) is the most common method in memory testing. Many BIST implementations have been proposed; they are based either on implementing a march element (ME), or on implementing a memory operation. However, these implementations suffer from a high area overhead and/or low flexibility and/or complex schemes such as prefetching and pipelining in order to perform at-speed testing. This study proposes a highly flexible and low area Memory BIST Engine able to perform at-speed testing without complex schemes. It is based on the Generic March Element (GME) Concept, and allows the specification of any suitable ME with any stress combination (e.g., address order, data-background) in an orthogonal way. The proposed BIST engine is capable of the following: (a) implements different test algorithms, both linear and non-linear, (b) uses different algorithm stresses, (c) easy extendable and/or modifiable, (d) requires a low area overhead, (e) performs at-speed testing, and (f) is generic for different memory size configurations.

Hardware was synthesized with the Synopsys Design Compiler with the Faraday UMC 90 nm Standard Process library. The results show an area overhead of 7.2 K (9 %) gates for a 16K x 16-bit memory at 500 MHz. Whereas, the state-of-the-art implementations reported area overheads varying from 8 K to 14 K gates at frequencies below 333 MHz. Moreover, the area overhead decreases from 9 to 0.01 % with increasing memory sizes from (16K to 16M) x 16-bit. The proposed BIST has a high coding efficiency; e.g., it requires only 36 bits to define March C+ algorithm, whereas previous studies require between 100 to 500 bits. Synthesizing for specific algorithm combinations can further reduce the area overhead and increase the operation frequency.

Faculty of Electrical Engineering, Mathematics and Computer Science

Generic and Orthogonal March Element based Memory BIST Engine

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Halil Kukner
born in Ankara, Turkey

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Generic and Orthogonal March Element based Memory BIST Engine

by Halil Kukner

Abstract

The International Technology Roadmap for Semiconductors (ITRS) forecasts that embedded memories will dominate the System-on-Chip (SoC) area by about 94% in no more than five years. In addition, the manufacturing process in the nano-era introduces new and complex failure mechanisms. The detection of such failures requires at-speed testing. Moreover, the complexity of electronic systems has reached a level where the accessibility of embedded memories using external test equipment is getting harder and costly, if not infeasible. Thus, high quality memory testing is crucial to improve the overall SoC quality.

Built-in Self-Test (BIST) is the most common method in memory testing. Many BIST implementations have been proposed; they are based either on implementing a march element (ME), or on implementing a memory operation. However, these implementations suffer from a high area overhead and/or low flexibility and/or complex schemes such as prefetching and pipelining in order to perform at-speed testing. This study proposes a highly flexible and low area Memory BIST Engine able to perform at-speed testing without complex schemes. It is based on the Generic March Element (GME) Concept, and allows the specification of any suitable ME with any stress combination (e.g., address order, data-background) in an orthogonal way. The proposed BIST engine is capable of the following: (a) implements different test algorithms, both linear and non-linear, (b) uses different algorithm stresses, (c) easy extendable and/or modifiable, (d) requires a low area overhead, (e) performs at-speed testing, and (f) is generic for different memory size configurations.

Hardware was synthesized with the Synopsys Design Compiler with the Faraday UMC 90 nm Standard Process library. The results show an area overhead of 7.2 K (9 %) gates for a 16K x 16-bit memory at 500 MHz. Whereas, the state-of-the-art implementations reported area overheads varying from 8 K to 14 K gates at frequencies below 333 MHz. Moreover, the area overhead decreases from 9 to 0.01 % with increasing memory sizes from (16K to 16M) x 16-bit. The proposed BIST has a high coding efficiency; e.g., it requires only 36 bits to define March C+ algorithm, whereas previous studies require between 100 to 500 bits. Synthesizing for specific algorithm combinations can further reduce the area overhead and increase the operation frequency.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-25

Committee Members :

Advisor:	Dr. Ir. Said Hamdioui, CE, TU Delft

I dedicate this thesis to my family, my friends, people from whom I learned, and to the reader.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank to my advisors Dr. Ir. Said Hamdioui and Prof. Dr. Ir. Ad van de Goor for their guidance, help, support, and fruitful discussions during this study. It was an honor for me and I was really lucky that I found a chance to work with these legendary academicians of the memory testing field.

Secondly, I would like to thank to my family for educating and growing me up, providing their support, showing their love, and always sharing their experience and knowledge.

Without my friends in Netherlands, it was impossible to be motivated and study. Hence I would like to thank to all of them, especially the ones that we spent a whole year in the Marcushof dorm. To mention, Fatih Abanoz, João Carreira, Hande Cetinay, Shuoxi Chen, Oktay Cikili, Gokturk Cinserin, Sibel Dumlu, Meliha Gozde Erdem, Sema Ermez, Onur Kaya, Miguel Miranda, Yunus Okmen, Berk Ozdirik, Doga Can Su Ozturk, Egemen Sahin, Caglayan Sandal, Bahar Yousefian, Emile Soulié, Busra Torgay, Ertan Umit, and all the friends in Delft. I really met with so many legendary people.

Lastly, I would like to present my gratitude and respect to all of our teachers, professors that gave their valuable time to educate and teach us something.

Halil Kukner
Delft, The Netherlands
November 28, 2010

Introduction 1

This chapter introduces some basics about memory testing and justifies its importance. It further summarize the main contribution of this work and provide an outline of the thesis.

This chapter is organized as follows. Section 1.1 starts with the background and motivation behind the Built-In Self-Test (BIST). Section 1.2 mentions the contributions of this work. Section 1.3 presents a brief outline of the thesis.

1.1 Background and Motivation

Here, the reasonings and increasing importance of memory testing are explained. Thereafter, the advantages and drawbacks of previous test approaches will be given.

1.1.1 Importance of memory testing

In relation to Moore's Laws, CMOS manufacturing process sizes are getting smaller: 65 nm in 2006, 45 nm in 2008 and today 32 nm in 2010, and that stream will further continue. However, this trend brings to electronic systems three major drawbacks: 1) lower chip yield, 2) more vulnerability to process, voltage and technology (PVT) variations, and 3) new fault mechanisms and more faults.

The CMOS scaling down trend brings more computing and processing power at higher frequencies with a lower area and power overhead. Higher computing power results in an increase of the data amount to-be processed. Thus electronic systems require larger memories. According to the International Technology Roadmap for Semiconductors (ITRS) forecasts [8], 94 % of a chip area will be dominated by the memory area in the following 4-5 years. In electronics production, system yield depends on the silicon area occupied by a chip. From the ITRS forecast, one can conclude that overall chip yield will highly depend on the memory quality.

Scaling down CMOS gate length increases the vulnerability of chip. Previous fault-free conditions will now start to cause faults such that the number of faults will increase.

In addition, new CMOS sizes will enable the designer to build new memory architectures. New memory architectures will bring their own fault mechanisms.

When those three effects (lower yield, higher vulnerability to PVT variations, new memory architectures) are combined, memories are most likely to contain more, old/new fault types with an increased number of appearances. Thus, memory testing is an important field where overall system quality and test time depend on.

1.1.2 Existing test approaches

Automated Test Equipment (ATE) was common in the testing world. However, as a result of scaling down CMOS sizes, ATE has the following drawbacks: 1) high cost, 2) lack of direct access to embedded memories, and 3) low quality testing.

Advances in technology brings a high variety of memory size configurations, number of input/output ports and signaling schemes (Double, Quad Data rates (DDR, QDR)). ATE test heads are designed as chip-specific. To be able to support so many different memory configurations, the cost of a test head is too high. Moreover, ATE requires periodical maintenance and upgrades to catch up with the scaling down technology trend.

Today, systems consist of several hundreds of distributed embedded memories in varying sizes, where a direct access from external to most of them are absent [30]. In some cases, accessing to those embedded memories is infeasible and uneconomical for an ATE.

In addition, in case of direct access, the path length from external to the to-be tested memory is too long to allow for a testing at the operational frequency of the memory (at-speed testing). In addition to the lack of at-speed testing, external access ports are generally shared, resulting in discontinuities of memory test operations (full-speed testing). Without at-speed testing, high quality test results are impossible.

Therefore, Automated Test Equipment (ATE) devices are becoming highly costly to support many memory configurations, chip specific test head design, and ineffective as a result of the lack of at-speed and full-speed testing, and lowered tester accuracies.

Those points above mandate a shift from ATEs to a Design-for-Test (DFT) based testing strategy. Solution is the Built-In Self-Test (BIST) concept.

BIST is a concept that enables a system to test itself. It may be an analog-mixed signal BIST, a logic BIST as well as a Memory BIST. This thesis focuses on the Memory BIST.

Compared to ATE, Memory BIST has the following advantages: 1) being low cost, 2) support of any memory configuration, 3) ease of memory access, and 4) at-speed testing.

BIST is produced as a part of the chip, thus it is a low cost solution. It does not require any maintenance or upgrade costs. There is no problem with supporting different memory configuration or memory access. In case of a high-speed BIST implementation, it operates at the same operational frequency with the memory.

On the other hand, BIST has some shortcomings, too: 1) it causes silicon area overhead, 2) requires some extra I/O pins to be controlled, 3) its own hardware requires self-testing, 4) requires an experienced BIST designer.

In addition to the mentioned advantages, BIST is significantly important as being the fundamental sub-block for a Built-In Self-Repair (BISR) system, too. Regarding to the reasons mentioned above, Figure 1.1 [124] shows that the yield rapidly decreases with increasing memory domination on the chip area. Without a proper repair strategy, it is impossible to catch higher yield levels. Therefore, BISR based yield enhancement is a rising trend due to its low cost, easy integration and in some cases being the only repair enabling solution. Before starting to repair, fault locations should be determined and a diagnosis procedure has to be performed with a BIST module.

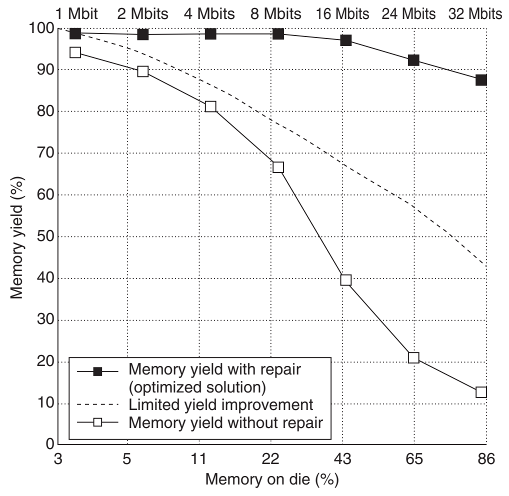


Figure 1.1: Memory yield optimization [124]

To conclude, mainstream DFT trend is heading towards the Built-In Self-Test, Diagnosis and Repair solutions.

1.2 Contribution of this thesis

This study targets a comprehensive and powerful BIST hardware solution with the following features: 1) high flexibility, 2) the algorithm stresses and the ME are in orthogonal to each other, 3) low area, 4) at-speed testing, 5) easy extend and/or modify, and 6) generic architecture for different memory configurations.

- High fault coverage: because of a highly flexible support of different memory test algorithm families (e.g., linear, non-linear), user-programmable and in-the-field programmable,

- Generic and Orthogonal March element based design: orthogonal generation of specified ME and algorithm stress combinations (i.e., address order, data value, addressing scheme, data background),

- Low area overhead: optimized hardware implementation, variable-length commands for an efficient command memory usage,

- At-speed testing: testing at the operational frequency of memory,

- Easily extend and/or modify: ease of adding new commands and stresses to hardware,

- Generic architecture: ease of realization for different memory size configurations,

1.3 Outline of the thesis

This thesis is organized as follows. Chapter 2 gives background information on memory architecture in a top down fashion; from behavioral memory to electrical model.

Chapter 3 defines the reduced functional memory model, and introduces the definition of fault primitives. A classification of the fault space will be presented, and thereafter a complete set of faults will be described.

Chapter 4 introduces the memory test notation and gives overview of several memory test algorithms from de-facto ones to fault primitive based developed test algorithms.

Chapter 5 reports the state-of-the-art of Memory BIST; it classifies different BIST implementations and discusses their advantages and drawbacks.

Chapter 6 proposes a Generic and Orthogonal March element based Memory BIST (GME MBIST) hardware. Command and register sets will be introduced. Later on, the implementation of several memory test algorithms written in GME MBIST test assembly language will be given.

Chapter 7 provides the low level architecture of GME MBIST Engine. RTL coding examples for address, data and control generations will be shown. Finally, the implementation results of GME MBIST hardware will be presented, discussed and compared with the state of the art Memory BISTs.

Chapter 8 is a user guide for the graphical user interface (GUI) of GME MBIST Engine. It shows an easy and fast utilization of GME MBIST hardware.

And lastly, Chapter 9 concludes the thesis with a summary of each chapter; it presents the major contributions of this thesis, and suggests a few points for further improvements and extensions.

2 Memory architecture

This chapter motivates the importance of using memory models. The existing different levels of abstraction will be briefly discussed in a top to bottom manner.

This chapter is organized as follows. Section 2.1 provides the reasonings behind the memory modeling. Section 2.2 explains the behavioral memory model; basic properties (e.g., pin configuration, control signaling) of a typical industrial SRAM will be shown. Section 2.3 presents the functionality of the memory operations. Section 2.4 explains the electrical memory level; sub-blocks (e.g., memory unit cell, address decoders) will be discussed in details.

2.1 Modeling memories

A memory test can be performed in two ways: physical real inspection of the device by removing the package, or applying some input test signals and checking for the consistency of memory with respect to a well-known system behavior. In physical testing, once the memory package is removed, the system is disturbed, even if fault-free chip is packaged again, still it might fail because of the latest packaging. Thus, physical inspection can be applied only for diagnostic purposes, not for testing or commercial marketing. The remaining option is the mainstream test strategy for semiconductor memories. Obviously, checking consistency is much easier, time and cost saving for mass production volumes.

For the consistency checking, a test procedure should be systematically developed and applied. However, developing an algorithm that takes into account all details and comprises the whole system as monolithic, is too complex, costly, and even if sometimes result-less; however, partitioning the system into smaller parts in a hierarchy decreasing from top to bottom, and then developing algorithms per each of them is an easier way. If an analogy should be given, one could think about the algorithmic complexities of hierarchical vs. flattened placement-routing steps of a typical ASIC flow. In short, a memory system is modeled in terms of sub-blocks for the simplification of the problem of ease of modeling and more precise fault localization. Memory cell arrays, address decoders, sense amplifiers, etc. are some of the memory sub-blocks, and they will be investigated in following sections.

One further step to simplify testing is increasing the level of faults. Meaning that physical faults should be modeled at a higher level. Physical diagnosis is quite hard and almost impossible due to the limited number of I/O pins for today's complex billions of transistor consisting ICs. Generally, there is not any direct connection from external world into a test candidate wire/gate/layer/physical location. Thus, those real-world faults should be modeled in such a way that the fault could be detectable/identified by examining the expected outputs after applying a deterministic input set to the system.

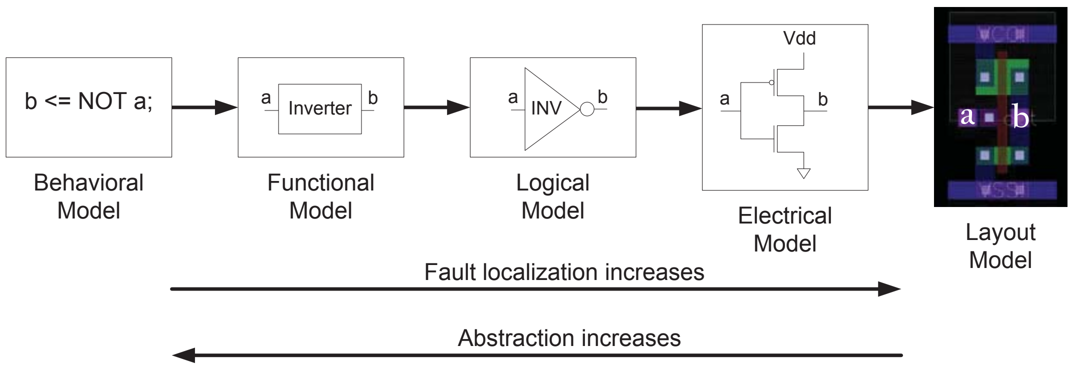


Figure 2.1: Memory models and abstraction levels

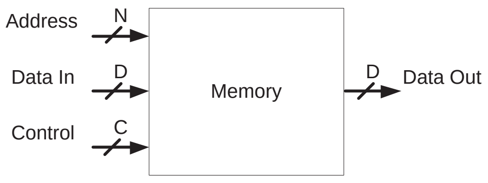


Figure 2.2: Memory behavioral model

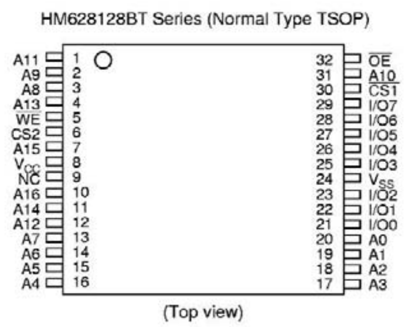
Figure 2.1 shows the levels of abstraction in memory modeling in a top to bottom manner; from behavioral to layout model. Increasing abstraction level, decreases the fault localization.

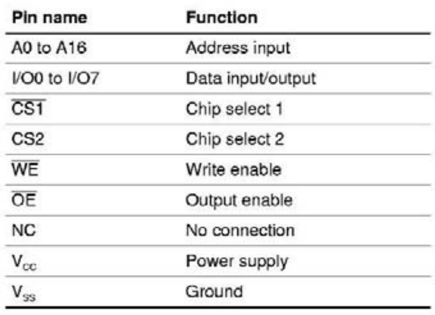
The behavioral model is the highest abstraction level that only gives the information on the input/output signals, and hides the internal details. **The functional model** is the level where the system is partitioned into specific sub-blocks. Each sub-block is responsible for a certain function. **The logical model** represents the system in terms of logic gates. Since a memory contains not only the logic gates but also the basic electrical components, this model is not commonly applied and will not be explained in this chapter. **The electrical model** presents the system in terms of the electrical components (e.g., transistors, resistors). It is frequently applied for the fault modeling and test algorithm development. **The layout model** is the lowest level where the geometrical drawings of the physical implementation are given. It visualizes the actual positions of transistors, wires and vias. This level is mostly applied for the fault diagnosis.

2.2 Behavioral memory model

As in Figure 2.2, this model considers the memory as a black box without supplying any information about internal memory structure. Sub-blocks, logic gates, wire connections, read/write circuitry, etc. are cloaked by the black-box, no details are given.

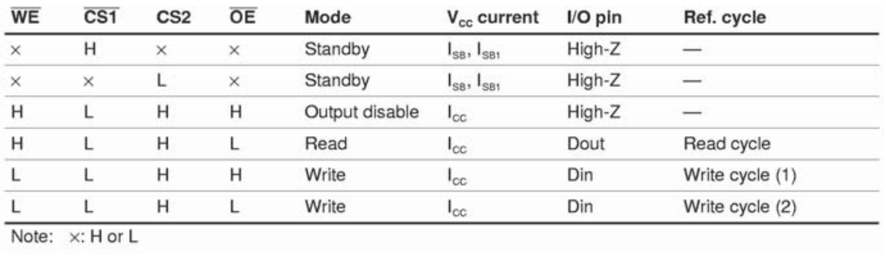
Operating conditions, AC/DC parameters, package information, pin configuration and names, truth table for control signals, timing diagrams and some application notes are provided to the end-user, see Figure 2.3, 2.4 and 2.5 [92]. Address, data-in/out,





Pin name	Function
A0 to A16	Address input
I/O0 to I/O7	Data input/output
$\overline{CS1}$	Chip select 1
CS2	Chip select 2
$\overline{WE}$	Write enable
$\overline{OE}$	Output enable
NC	No connection
V_{CC}	Power supply
V_{SS}	Ground

Figure 2.3: Hitachi 1 Mb (512 rows, 256 columns, 8-b data) SRAM pin configuration and description [92]



$\overline{WE}$	$\overline{CS1}$	CS2	$\overline{OE}$	Mode	V_{cc} current	I/O pin	Ref. cycle
×	H	×	×	Standby	I_{SB}, I_{SB1}	High-Z	—
×	×	L	×	Standby	I_{SB}, I_{SB1}	High-Z	—
H	L	H	H	Output disable	I_{CC}	High-Z	—
H	L	H	L	Read	I_{CC}	Dout	Read cycle
L	L	H	H	Write	I_{CC}	Din	Write cycle (1)
L	L	H	L	Write	I_{CC}	Din	Write cycle (2)

Note: ×: H or L

Figure 2.4: Functional diagram for the SRAM control signals [92]

read/write enabling and several control signals are presented. To be able to operate a memory correctly, the user should apply the required signal values at the right time intervals. At the behavioral model, no extra information on the memory internal architecture, logic technology (CMOS, TTL, ECL, etc.) or process sizes are required to operate it.

In Figure 2.4 [92], three main control signals for a SRAM and their logic states are shown to perform a read/write operation. For a read operation, chip should be selected by low asserting $\overline{CS1}$, output enable should be activated by low asserting $\overline{OE}$ and write enable should be deactivated by high asserting $\overline{WE}$. For a write operation, the chip should be selected, write enable is activated by low asserting $\overline{WE}$, and output enable is deactivated by high asserting $\overline{OE}$. $\overline{CS2}$ is an extra chip select signal for power standby mode by disabling clock signal. Figure 2.5 shows the above explanation in signal waveforms.

2.3 Functional memory model

Memory cell array, row and column address decoders, read/write circuitry, input/output data registers and control logic are the main sub-parts in a functional memory model as shown in Figure 2.6.

For a read operation, after applying certain control signals, the address of the requested memory bit/word is sent to memory; decoded to select the required row and

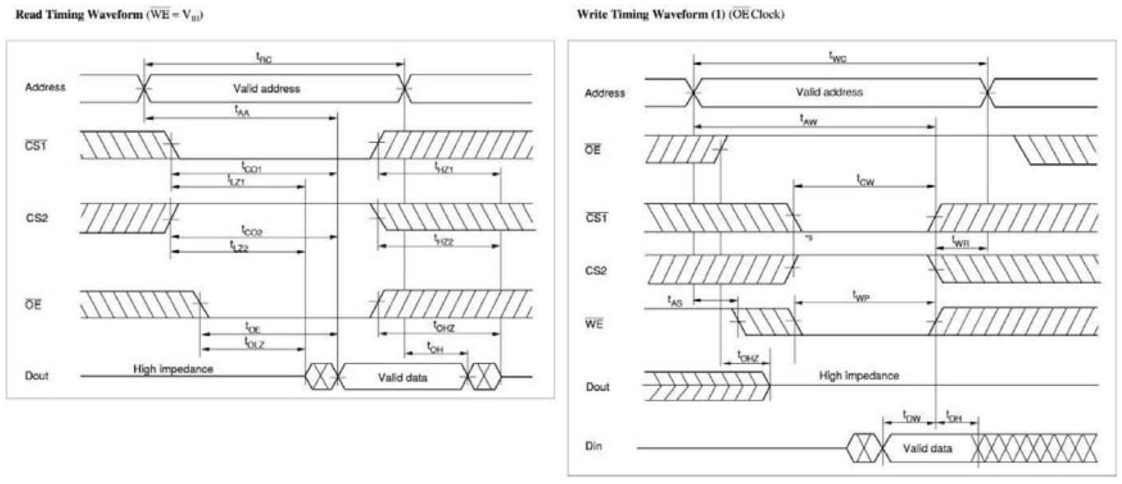


Figure 2.5: SRAM read and write timing diagrams [92]

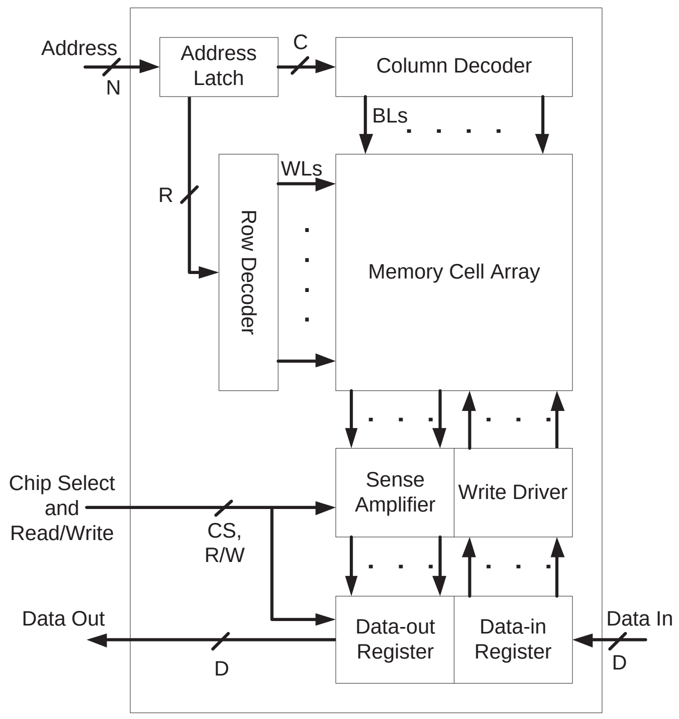


Figure 2.6: Functional model of a single-port SRAM

columns. Bit/word data is amplified by the sense amplifiers and finally registered by the data register module. Reversely, to perform a write operation, data is loaded into the data register and written into the cells by the write driver. The data lines going in/out from the data register are generally combined as bidirectional lines to decrease the number of pins and the cost of production.

Address bits are separated into two parts: lower and higher parts. Generally, lower

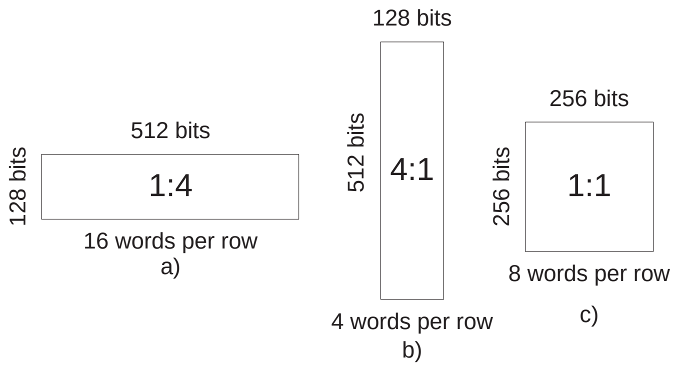


Figure 2.7: Several aspect ratios for a 64 Kb memory array with 32-bit data word: a) 1:4, b) 4:1, c) 1:1

parts go to the column decoder, while higher bits to the row decoder. After decoding, 2^R rows and 2^C columns are able to be selected. A memory cell array consists of $2^R * 2^C * D$ memory unit cells. In a memory, the total number of columns, $2^C * D$ is a multiple integer (1, 2, ... etc.) of D bit data inputs. Whereas, the total number of rows is a designer's choice, traditionally chosen as a power of 2, 2^R, not to waste the decoding logic and to avoid unintended selection of any row [88].

The *memory array aspect ratio* is the ratio of the memory array length to its width. For example, a 64 Kb array with a 32-bit data words could be designed in possible ways as in Figure 2.7. The aspect ratio is an indicator of memory dimensions and decoder sizes. An n-bit memory can be minimized to $\sqrt{n}$ in two dimensions. Thus, decoder area could be decreased from n to $2\sqrt{n}$ [100].

Several sub-modules could vary between memory types, for example refresh logic in DRAMs does not exist in SRAMs.

2.4 Electrical memory model

In this section, electrical properties of main sub-blocks: memory cell array, row/column address decoders, read/write circuits from the Figure 2.6 will be explained from an electrical perspective. It is hoped that memory faults and test algorithms at the following chapters, will be better understood and mapped to this background information.

2.4.1 Memory cell array

The main building structure of a memory is the memory cell. An SRAM memory cell has a bistable circuit characteristic. Meaning that it can be driven into one of the two states and retains its state as the power goes on.

The *beta ratio, (β)* is an indicator of the cell stability. It is the ratio between the strengths of pull-down transistor to pass-transistor, [2]:

$$\beta = \frac{W_{eff}PullDownFET/L_{eff}PullDownFET}{W_{eff}PassFET/L_{eff}PassFET}$$

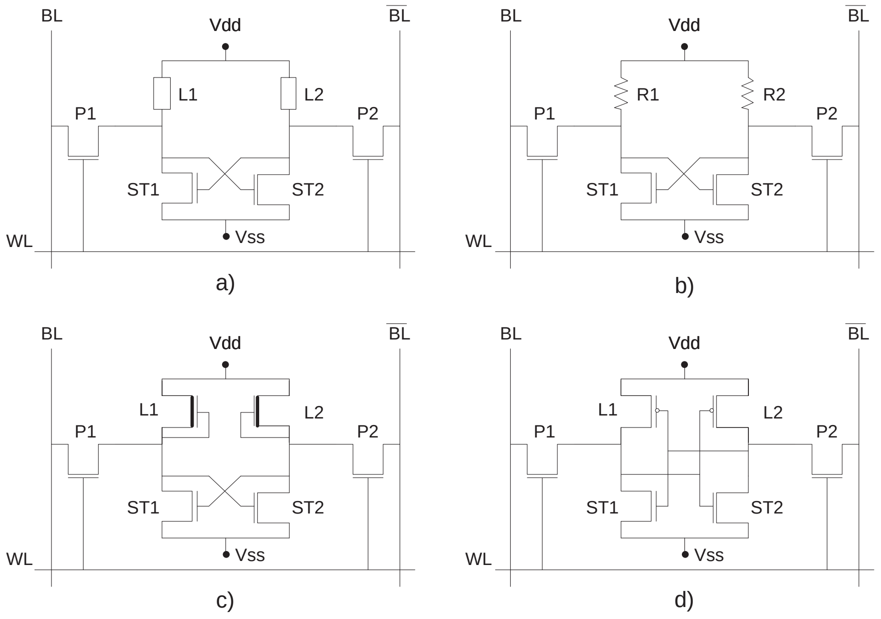


Figure 2.8: a) General SRAM unit memory cell and various realizations: b) 4T with resistor load, c) 6T with depletion mode NMOS load, d) 6T CMOS

Strength implies the effective dimensions, not the drawn ones on layout. Generally, beta ratio is chosen between 1.5 to 2.0 to have a non-destructive read operation. Otherwise, when it is below 1.0, read activity disturbs the data stored in the cell [2] such that the SRAM read operation is not non-destructive, anymore.

The design of a memory cell varies with the memory application area, design choices (quasi-static, sub-threshold, etc.) or logic technology (CMOS, TTL, ETL, etc.) as in Figure 2.8. Mainly, a cell consists of two cross-coupled inverting storage elements (ST1 and ST2), two load elements (L1 and L2) and two access elements (P1 and P2) as shown in Figure 2.8 a). Storage of the information is achieved by a latch which is formed by two cross-coupled inverters. The data stored at the drains of storage elements (ST1 and ST2) are read/written through the pass transistors (P1 and P2).

Accessing a memory cell is performed via *the Word Line (WL)* and *bit lines (BL) and* $\overline{BL}$. One WL passes through the gates of all pass transistors in a row, and is controlled by the row address decoder. Similarly, two complementary BLs per cell are connected to the source/drain of the pass transistors, and controlled by the column address decoder.

To write data, the BLs are loaded with complementary logic values (‘1’ and ‘0’), afterwards the WL is pulled up to a high logic level to enable transition through the pass transistors. This signaling order is important, because the storage elements of a cell are weaker than the driving force of BLs. Finally, the new data value is conserved bistably at the drains of storage and load elements, see Figure 2.8a).

In a read operation, first both BLs are precharged to logic HIGH, then the WL is pulled up. Once the cell is accessed, the stored data lowers the voltage value in one of the BLs, while the other BL is not affected. For example, when a ‘0’ stored cell is read,

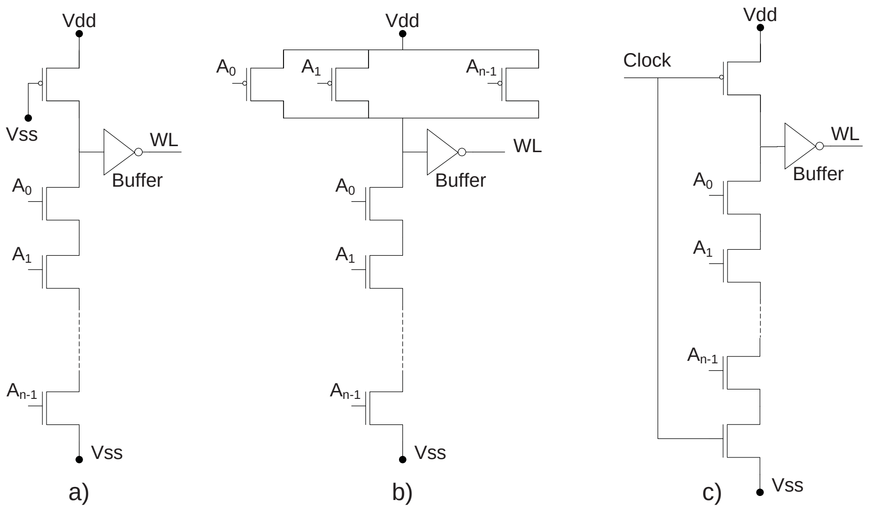


Figure 2.9: a) PMOS-load and b) CMOS styles static row decoders, c) dynamic row decoder [36]

the true bit line is discharged while the complement bit line remains high. In reverse, when a '1' stored cell is read, the complement bit line is discharged and the true bit line stays high. This discharge at one of the bit lines is only a small amount in the range of 100 mV [2], and is amplified and converted to a logic value by the sense amplifier. Afterwards, it is registered by data register in Figure 2.6.

As mentioned before, the internal structure of a cell can vary due to the logic technology. In Figure 2.8 b), the SRAM cell with resistor (polysilicon) load elements is shown. This design choice reduces the silicon area, however increases dissipated power due to the continuous flowing current through resistors. Another choice is using depletion-mode NMOSs instead of polysilicon resistors. This design occupies a higher silicon area with a decreased power consumption compared to previous design with polysilicon load but still higher than the CMOS design style, Figure 2.8 c). Finally, the mostly accepted and dominating design choice in today's IC market is 6T CMOS SRAM unit cell, Figure 2.8 d). Static power consumption is minimized and, except for the leakage current, it only dissipates dynamic power during the switching activity. Obviously, the CMOS process steps are more complex and costly than the previous design alternative.

2.4.2 Row/Column address decoders

For row decoders, both static or dynamic design styles might be adapted. A static row decoder can be implemented with a NOR, AND (NAND with a inverter at its output) or NAND gates. Figure 2.9 b) [36] shows an AND based implementation. Address bits A0 to A_{n-1} or their complements are applied to the gates to choose the correct word line. For example, to chose the word line for the row address 13 (00001101) for Figure 2.9 b) AND style, applied addressing signals should be as a7-a0: LLLLHHLH (L: Low, H: High asserted). In AND style, address inputs at the gates should be low asserted.

Upper network might be implemented with a PMOS-load, see Figure 2.9 a), to de-

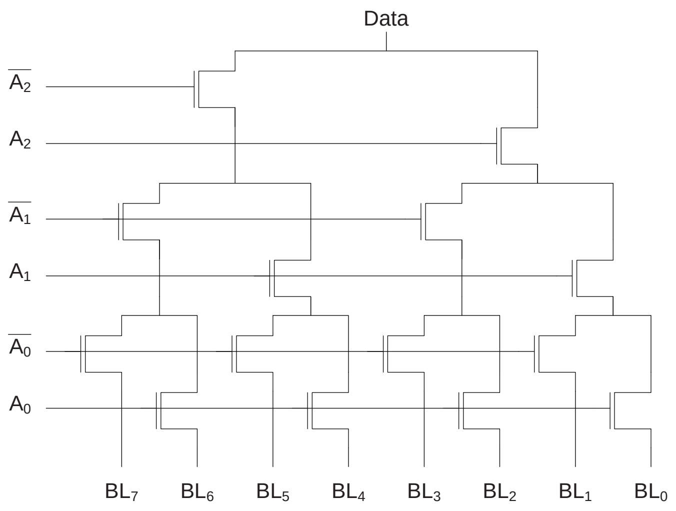


Figure 2.10: Logarithmic tree column decoder

crease the decoder area and speeding up the device due to the halfened capacitance load at the output. However, this choice causes an increased leakage current and power dissipation that are resulting in a shorter device life. Similarly, NMOS-load could be put to the bottom network in AND style.

Figure 2.9 c) [36] shows a dynamic row decoder as another design choice. Since it does not require the complete input set at the upper network, it can be build up with fewer transistors. A dynamic row decoder uses clocking, thus consumes less power only while conducting, however clock signal should be routed to the decoders.

For the column decoder, a logarithmic tree decoder example will be given. It is a single output decoder with $\log_2 n$ levels and need to route $2.\log_2 n$ addressing signals, however, data-in BLs should propagate several levels to be read/written, resulting in a slow response. Figure 2.10 shows a logarithmic tree column decoder accessing 8 BLs by decoding 3-bit column address. Transmission gates could be used to build fast column decoders. Data only propagate over the transmission gates and arrives to the sense amplifier or comes from the write-in circuit. But timing and controlling should be handled carefully, because several cells could drive the same wire which can cause hazards.

2.4.3 Read/Write circuits

There are different sub-blocks in this part: precharge circuit, isolation circuit, sense amplifier and write driver.

Precharge Circuit

Charge-up of the BLs before the read operation is done by *the Precharge Circuit.* Figure 2.11 a) shows the 3 transistor structure. Two bottom PFETs pull up the BLs to Vdd level. The upper PFET is generally optional and added to equalize the potentials of bit line couples, see .

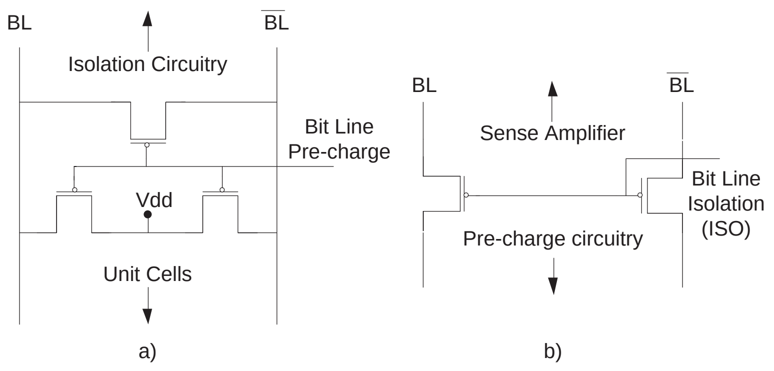


Figure 2.11: a) Precharge and b) isolation circuits

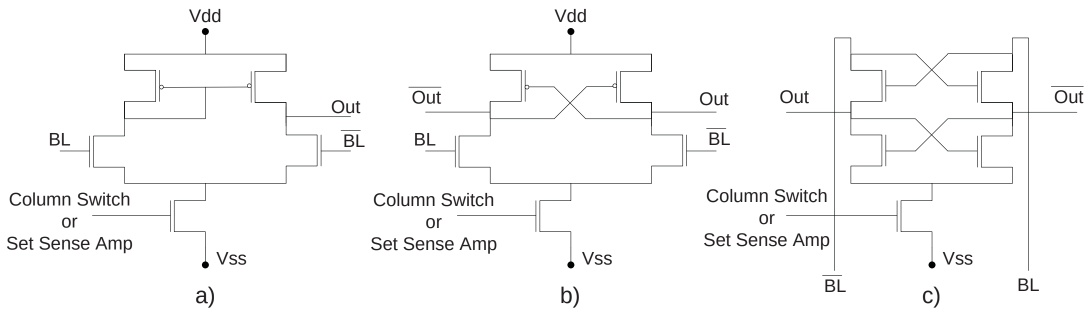


Figure 2.12: a) single-ended and b) differential-ended voltage mode (latch type) sense amplifiers [36, 2], c) Current mode read circuit [2]

Isolation Circuit

After the precharge circuit, to speed up the read operation, BLs are isolated from the sense amplifier by *the Isolation Circuit*, see Figure 2.11 b). The reason of usage is the capacitive load due to the connection of lots of cells to BLs and long metalization of BLs.

Sense Amplifier

The read circuit is called the sense amplifier. There are several implementations for sense amplifiers varying from single-ended to differential, see Figure 2.12 a) and b) , or voltage mode (latch) to current mode as in Figure 2.12 c). For the voltage mode read circuits, differential sense amplifiers are more preferred for a high performance SRAMs due to their fast switching capability to sense and convert small voltage differences on BLs. Differential-ended sense amplifiers are faster than their single-ended counterparts due to their cross-coupled inner structure. The sense amplifier is activated by *the Column Switch (CS)* or *Set Sense Amplifier (SSA)* signal.

A current type sense amplifier, Figure 2.12 c), determines the stored data value in the unit cell depending on the current flowing in BLs. A current type sense amplifier has

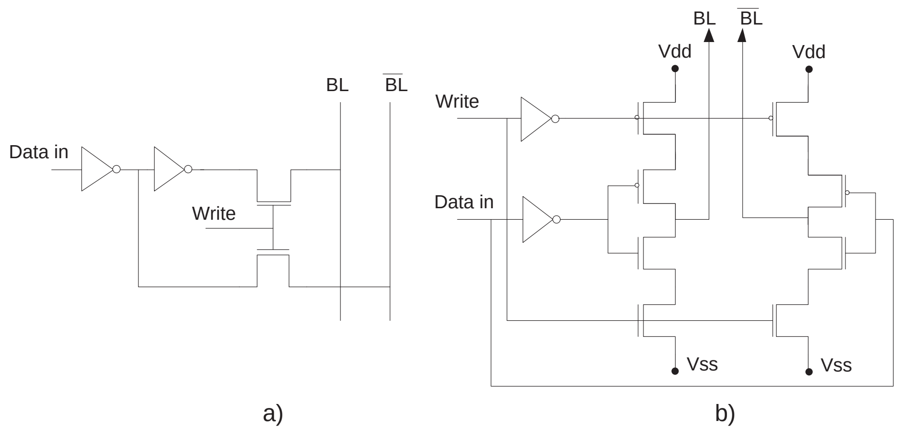


Figure 2.13: Write drivers: a) pass transistor based [36], b) gated inverter based [2]

a different characteristic than the voltage type sense amplifier. In voltage type, when the sense amplifier is activated, it is locked into the data stored in cell. Further, changes on voltage level in BLs do not reflect to the sense amplifier output. However, current type amplifier corrects its output when signal values change in BLs. For an example case, let us say: the signals at the beginning conditions were erroneous, and after a while they changed back to their true value. If a voltage type read circuitry is being applied, then the output would be the first faulty value.

Write Driver

The transfer elements of an SRAM cell are NFETs which have an effective driving capability for a logic '0', but not for '1'. Thus, the write operation is performed by writing a '0' either into BL or $\overline{BL}$. Then, cross-coupled inverters inside the cell transform '0' to a logic '1' at the other storage node. When the write signal goes high, the data is delivered to the BLs. Since, loading a '0' into one of the BLs is the main purpose, PFETs and NFETs may be sized same, not as in traditional way, 2:1, see Figure 2.13 b) [2].

For further information on SRAM architectures, transistor sizing, unit cell layouts, BL to WL or BL to BL couplings, power dissipation on memories, SRAM implementation with new logic technologies, etc. please refer to [88, 69, 83, 61].

Memory faults 3

This chapter introduces the reduced functional memory model and the concept of the fault primitive. In addition, static and dynamic fault spaces are given.

This chapter is organized as follows. Section 3.1 simplifies the functional memory model by reducing the number of sub-blocks to three: memory cell array, address decoders and peripheral circuits. Section 3.2 provides the definition of the fault primitive concept, and classifies them. Section 3.3 and 3.4 derive the complete set of static and dynamic fault primitives, and group them under the functional fault models.

3.1 Reduced functional memory model

To derive the functional fault models, a reduced functional memory model is build, see Figure 3.1. It is the simplified version of the general functional memory model from the Chapter 2, by leaving behind only those 3 sub-modules:

1. memory cell array,
2. address decoder consisting row/column decoders and address registers,
3. peripheral circuitry consisting rest of the memory such as read/write circuitry (write driver, sense amplifier, data registers) and pre-charge circuits.

Once, the memory model is simplified, fault modeling will be more straightforward and time-saving. Furthermore, analyses at the end will be easier, faster, point-specific.

3.2 Concept and classification of fault primitives

To understand whether a memory component is faulty or not, a number of memory operations should be applied to the memory-under-test, and later on the memory read-outs should be compared with expected responses. Thus, in a fault model, there should be two essential elements:

1. a list of memory operations that sensitizes the fault, called as the *sensitizing operation sequence (SOS)*,
2. a list of mismatches between expected and observed behaviors, called as the *faulty behavior.*

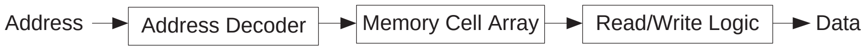


Figure 3.1: Reduced functional memory model [100]

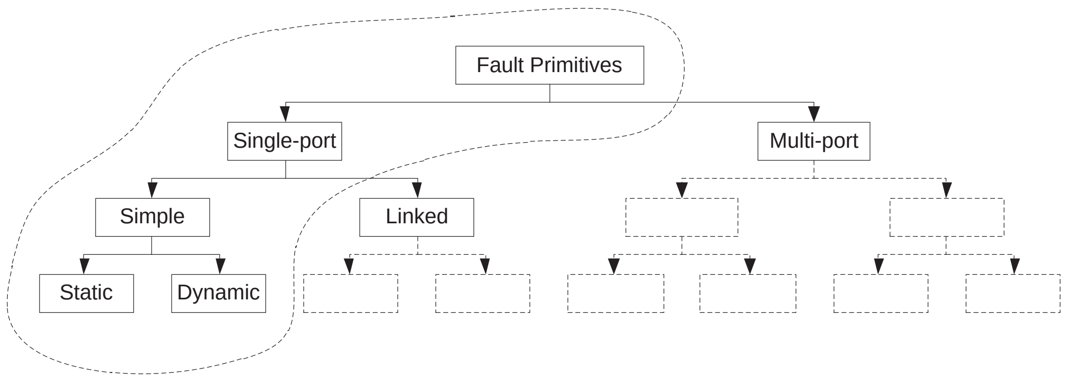


Figure 3.2: Family of fault primitives [36]

In addition to the two essential elements, the read-out value corresponds to several read-related faults and also relevant for a fault model. Thus, a *Fault Primitive (FP)* consists of three elements in an order, and annotated as $< S/F/R >$ [105]. S is the sensitizing operation sequence, F is the faulty cell behavior, and R is the read operation output. When a number of FPs are combined, a *functional fault model (FFM)* is formed.

Once we have the $< S/F/R >$ definition above, all of the faulty behavior combinations can be generated mathematically by using combinations of them. Figure 3.2 [36] shows the 6 main FP categories: single-/multi-port, simple/linked and static/dynamic FPs.

Simple faults do not affect the behavior of any other faults, whereas *linked faults* can change each other's behavior and can cause fault masking. Basically, linked faults consist of two or more simple faults. A *static fault* is sensitized by at most one operation; i.e., the number of different operations in their SOS should be less or equal than 1, ($\#O \leqslant 1$). Reversely, *dynamic faults* are only sensitized by more than one sequential operation, ($\#O > 1$). *Single-port faults (1PFs)* are sensitized by at most one port access, ($\#P \leqslant 1$) while *multi-port faults (nPFs)* require more than one port operations simultaneously. 1PFs can exist both in single and multi port memories. *Single-cell faults* occur in the same cell where the SOS is applied, whereas in *multi-cell faults*, the sensitized cell is different from the effected cell(s). When two cells are involved, it is named a two-cell fault. In case of more than two cells, it can be generalized as k-cell faults. A brief description will be given in the coming sections of neighborhood pattern sensitive faults.

The dashed line in Figure 3.2 [36] shows the focus of this chapter: single-port simple static and dynamic FPs will be discussed in the following sections.

3.3 Static faults

Static faults are classified into 3 main classes: memory cell array, address decoder and peripheral circuit faults.

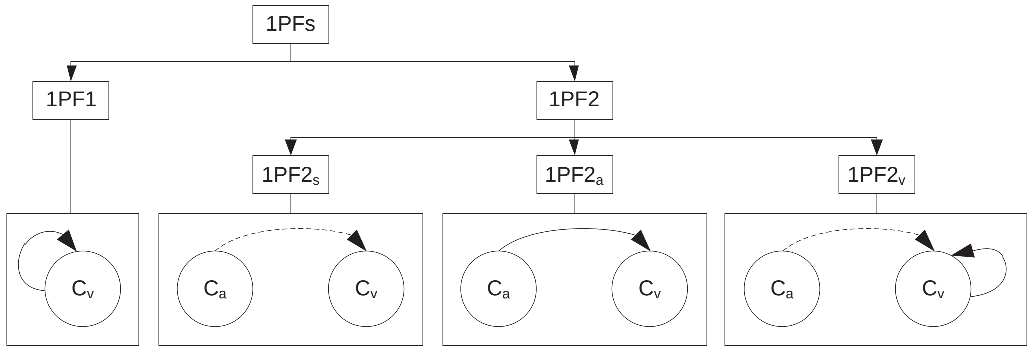


Figure 3.3: Class of 1PFs [36]

3.3.1 Static memory cell array faults (sMCAFs)

Before describing fault primitives, a short overview of fault types will be given here. Generally, stuck-at faults *(SAF)* are common faults in SRAMs. In case of a SAF, a cell cannot be overwritten with its complementary data. They are generally the result of shorts between interconnects such as a transistor short to supply voltage or ground level. A cell in a SAF is named as a SA0 or SA1. In a stuck-open fault *(SOF)*, a cell cannot be accessed because of an open access line; word line, bit line, etc. In a transition fault *(TF)*, a cell can perform either a "0" to "1" or an "1" to "0" transition, but not both of them. A TF can be caused by an absent access transistor. In a coupling fault *(CF)*, two or more cells are coupled to each other. It is observed when a performed transition on an *aggressor cell (a-cell c_a)* affects the contents of a *victim cell (v-cell c_v)*. In a bridging fault *(BF)*, two or more lines are bridged by a physical existence of any material (dust, circuit process material residues, etc.). Generally, it has a bidirectional behavior. In a state coupling fault *(SCF)*, the state of the v-cell is affected by the state of an a-cell rather than its transition. In a data retention fault *(DRF)*, a cell loses its data after a certain time, although power of memory is still on. It is mainly due to the leakages and DRFs are becoming a greater issue with the scaling down CMOS process technology.

Single-port faults have two sub-groups: single-cell *(1PF1)* and multi-cell *(1PFn)* FPs. In this chapter, only the single and two-cell FPs will be examined in detail.

As shown in the Figure 3.3 [36], when the sensitized cell and fault appeared cell are same, then it is a 1PF1. When two cells are involved, it a 1PF2. 1PF2 can be categorized under one of those 3 groups:

1. $1PF2_s$: Not an operation but the state of the a-cell sensitizes a fault in the v-cell.
2. $1PF2_a$: A single-port operation to the a-cell sensitizes a fault in the v-cell.
3. $1PF2_v$: A single-port operation to v-cell when the a-cell is in an exact state, sensitizes a fault in v-cell.

3.3.1.1 Single-cell fault primitives

In the c single-cell fault primitive notation, S, sensitizing state or operation, can be an element of $\{0, 1, r0, r1, 0w0, 1w1, 0w1, 1w0\}$. F, faulty cell behavior, can be an element of

Table 3.1: The complete set of 1PF1 FPs [36]

#	S	F	R	$< S/F/R >$	FFM	#	S	F	R	$< S/F/R >$	FFM
1	0	1	-	$< 0/1/- >$	SF	2	0	?	-	$< 0/?/- >$	USF
3	1	0	-	$< 1/0/- >$	SF	4	1	?	-	$< 1/?/- >$	USF
5	$0w0$	1	-	$< 0w0/1/- >$	WDF	6	$0w0$	?	-	$< 0w0/?/- >$	UWF
7	$1w1$	0	-	$< 1w1/0/- >$	WDF	8	$1w1$	?	-	$< 1w1/?/- >$	UWF
9	$0w1$	0	-	$< 0w1/0/- >$	TF	10	$0w1$	?	-	$< 0w1/?/- >$	UWF
11	$1w0$	1	-	$< 1w0/1/- >$	TF	12	$1w0$	?	-	$< 1w0/?/- >$	UWF
13	$0r0$	0	1	$< 0r0/0/1 >$	IRF	14	$0r0$	0	?	$< 0r0/0/? >$	RRF
15	$0r0$	1	0	$< 0r0/1/0 >$	DRDF	16	$0r0$	1	1	$< 0r0/1/1 >$	RDF
17	$0r0$	1	?	$< 0r0/1/? >$	RRDF	18	$0r0$	?	0	$< 0r0/?/0 >$	URF
19	$0r0$	?	1	$< 0r0/?/1 >$	URF	20	$0r0$	?	?	$< 0r0/?/? >$	URF
21	$1r1$	1	0	$< 1r1/1/0 >$	IRF	22	$1r1$	1	?	$< 1r1/1/? >$	RRF
23	$1r1$	0	0	$< 1r1/0/0 >$	RDF	24	$1r1$	0	1	$< 1r1/0/1 >$	DRDF
25	$1r1$	0	?	$< 1r1/0/? >$	RRDF	26	$1r1$	?	0	$< 1r1/?/0 >$	URF
27	$1r1$	?	1	$< 1r1/?/1 >$	URF	28	$1r1$	?	?	$< 1r1/?/? >$	URF

$\{0, 1, \uparrow, \downarrow, ?\}$. $\uparrow$ / $\downarrow$ are up/down transitions and ? shows that state of the cell is undefined when the voltage of data storage nodes in the cell are too close. R, read output of SRAM, can be an element of $\{0, 1, ?, -\}$. ? is for a random logic value when the voltage mismatch between BLs are too small. '-' means no read operation is performed and used when S is an write operation or a state value. Number of S.F.R combinations that are obtained by those clusters are 36(28 faulty + (2+4+2) non-faulty), however after the elimination of non-faulty combinations (i.e., $< 0/0/- >$, $< 0w1/1/- >$, $< r0/0/0 >$, etc.), 28 FPs of 1PF1 are left. The complete set can be seen in Table 3.1 [36].

This complete set is generated as:

- for $S \in \{0, 1\}$
  when $S = 0$, $F \in \{1, ?\}$ and $R = -$; as in FP1 and FP2.
  when $S = 1$, $F \in \{0, ?\}$ and $R = -$; as in FP3 and FP4.

- for $S \in \{0w0, 1w1, 0w1, 1w0\}$
  when $S = 0w0$, $F \in \{\uparrow, ?\}$ and $R = -$; as in FP5 and FP6.
  when $S = 1w1$, $F \in \{\downarrow, ?\}$ and $R = -$; as in FP7 and FP8.
  when $S = 0w1$, $F \in \{0, ?\}$ and $R = -$; as in FP9 and FP10.
  when $S = 1w0$, $F \in \{1, ?\}$ and $R = -$; as in FP11 and FP12.

- for $S \in \{r0, r1\}$
  when $S = r0$
    when $F = 0$ and $R \in \{1, ?\}$; as in FP13 and FP14.
    when $F \in \{\uparrow, ?\}$ and $R \in \{0, 1, ?\}$; as in FP15 to FP20.
  when $S = r1$
    when $F = 1$ and $R \in \{0, ?\}$; as in FP21 and FP22.
    when $F \in \{\downarrow, ?\}$ and $R \in \{0, 1, ?\}$; as in FP23 to FP28.

3.3.1.2 Single-cell functional fault models

In this section, above generated FPs are grouped into FFMs. FFM names and grouping the FPs have some historical roots.

1. **State Fault (SF):** Without accessing a cell, if the data inside that cell reverses, then this is called as a *state fault.* SF does not require any operation, is only related to the initial value of the cell. It has 2 FPs: $< 0/1/- >$ and $< 1/0/- >$.

2. **Undefined State Fault (USF):** Similar to SF, if a cell changes its state from a defined to an undefined state without any access into, then this is called as an *undefined state fault* and has 2 FPs: $< 0/?/- >$ and $< 1/?/- >$.

3. **Write Destructive Fault (WDF):** If a non-transition write operation reverses the data in the cell, than this is called as a *write destructive fault* and has 2 FPs: $< 0w0/\uparrow/- >$ and $< 1w1/\downarrow/- >$.

4. **Undefined Write Fault (UWF):** If a write operation moves the cell into an undefined state, than this is called as an *undefined write fault* and has 4 FPs: $< 0w0/?/- >$, $< 1w1/?/- >$, $< 0w1/?/- >$ and $< 1w0/?/- >$.

5. **Transition Fault (TF):** If a cell can not succeed to store the data in a transition write operation, then this is called as a *transition fault* and has 2 FPs: $< 0w1/0/- >$ and $< 1w0/1/- >$.

6. **Incorrect Read Fault (IRF):** In a read operation, if the read data returning from SRAM is wrong, however the data stored in the cell is still correct, then this is called as an *incorrect read fault* and has 2 FPs: $< r0/0/1 >$ and $< r1/1/0 >$.

7. **Random Read Fault (RRF):** In a read operation, if the read data returning from SRAM is random, however the data stored in the cell is still correct, then this is called as a *random read fault* and has 2 FPs: $< r0/0/? >$ and $< r1/1/? >$.

8. **Deceptive Read Destructive Fault (DRDF):** In a read operation, if the read data returning from SRAM is correct, however the data stored in the cell is reversed, then this is called as a *deceptive read destructive fault* and has 2 FPs: $< r0/\uparrow/0 >$ and $< r1/\downarrow/1 >$.

9. **Read Destructive Fault (RDF):** In a read operation, if the read data returning from SRAM is incorrect, and the data stored in the cell is reversed, then this is called as a *read destructive fault* and has 2 FPs: $< r0/\uparrow/1 >$ and $< r1/\downarrow/0 >$.

10. **Random Read Destructive Fault (RRDF):** In a read operation, if the read data returning from SRAM is random and the data stored in the cell is reversed, then this is called as a *random read destructive fault* and has 2 FPs: $< r0/\uparrow/? >$ and $< r1/\downarrow/? >$.

11. **Undefined Read Fault (URF):** If a read operation puts the cell into an undefined state, than this is called as an *undefined read fault.* The data returning from

SRAM can be either correct, wrong or random. URF has 6 FPs: $< r0/?/0 >$, $< r0/?/1 >$, $< r0/?/? >$, $< r1/?/0 >$, $< r1/?/1 >$ and $< r1/?/? >$.

12. **Stuck-at Fault (SAF):** Whatever operation is applied to the cell, if it is stuck-at a logic value, then this is called as a *stuck-at fault* and has 2 FPs: $< \forall/0/- >$ and $< \forall/1/- >$. $< \forall/0/- >$ is equal to any of $< 1/0/- >$, $< 0w1/0/- >$, $< 1w1/0/- >$ and the cell has the faulty behavior of those 3 FPs. Same for the $< \forall/1/- >$.

13. **No Access Fault (NAF):** When a cell does not have any access, the stored data can not be changed and the read data returns randomly, then this is called as a *no access fault* and has 4 FPs: $< 0w1/0/- >$, $< 1w0/1/- >$, $< r0/0/? >$ and $< r1/1/? >$.

14. **Data Retention Fault (DRF):** Without any access, if the cell reverses its data after a certain time T, then this is called as a *data retention fault* and has 4 FPs: cell is unable to store the logic 1 and switches to 0 after time T: $< 1_T/\downarrow/- >$, and similarly for the rest: $< 0_T/\uparrow/- >$, $< 0_T/?/- >$ and $< 1_T/?/- >$.

3.3.1.3 Two-cell fault primitives

$< S_a; S_v/F/R >$ is the notation for 1PF2s [36]. 1PF2s have 3 types as seen in Figure 3.3: $1PF2_s, 1PF2_a, 1PF2_v$. S_a and S_v are the states/sensitizing sequences for a-cell and v-cell, respectively. Set of S_i is defined as: $S_i \in \{0, 1, 0w0, 1w1, 0w1, 1w0, r0, r1\}$ where $(i \in \{a, v\})$, $F \in \{0, 1, \uparrow, \downarrow, ?\}$ and $R \in \{0, 1, ?, -\}$. The complete set has 80 FPs (8 $1PF2_s$s, 48 $1PF2_v$s, 24 $1PF2_a$s) and can be seen in Table 3.2 [36].

To refresh, those FPs are caused by simple, static (at most one operation, $\#O \leqslant 1$) and single-port (1P) operations. Then complete set of 1PF2 FPs are generated as:

- for $S_a \in \{0, 1\}$: Notation is $< x; S_v/F/R >$ where $x \in \{0, 1\}$. There are 56 FPs in total, 28 for $< 0; S_v/F/R >$ and 28 for $< 1; S_v/F/R >$.
  - when $S_v \in \{0, 1\}$: state of the a-cell sensitizes a fault in the v-cell, case $1PF2_s$; generates 8 FPs as in FP1 to FP4.
  - when $S_v \in \{0w0, 1w1, 0w1, 1w0, r0, r1\}$: operation on the v-cell with a certain state in the a-cell sensitizes a fault in the v-cell, case $1PF2_v$; generates 48 FPs as in FP5 and FP28.
- for $S_a \in \{0w0, 1w1, 0w1, 1w0, r0, r1\}$, $S_v \in \{0, 1\}$ and $R = -$: Notation is in the form of $< S_a; x/F/- >$ where $x \in \{0, 1\}$. There are 24 FPs, 12 per each case of S_v. Operation on the a-cell sensitizes a fault in the v-cell, case $1PF2_a$.
  - when $S_v = 0$ and $F \in \{\uparrow, ?\}$; as in FP29 to FP40.
  - when $S_v = 1$ and $F \in \{\downarrow, ?\}$; as in FP41 to FP52.

3.3.1.4 Two-cell functional fault models

80 1PF2s are grouped in historically determined several FFMs under three main classes:

Table 3.2: The complete set of 1PF2 FPs; $x \in \{0,1\}$ [36]

#	S_a	S_v	F	R	$< S_a, S_v/F/R >$	FFM	#	S_a	S_v	F	R	$< S_a, S_v/F/R >$	FFM
1	x	0	1	-	$< x; 0/1/- >$	CFst	2	x	0	?	-	$< x; 0/?/- >$	CFus
3	x	1	0	-	$< x; 1/0/- >$	CFst	4	x	1	?	-	$< x; 1/?/- >$	CFus
5	x	$0w0$	1	-	$< x; 0w0/1/- >$	CFwd	6	x	$0w0$	?	-	$< x; 0w0/?/- >$	CFuw
7	x	$1w1$	0	-	$< x; 1w1/0/- >$	CFwd	8	x	$1w1$	?	-	$< x; 1w1/?/- >$	CFuw
9	x	$0w1$	0	-	$< x; 0w1/0/- >$	CFtr	10	x	$0w1$	?	-	$< x; 0w1/?/- >$	CFuw
11	x	$1w0$	1	-	$< x; 1w0/1/- >$	CFtr	12	x	$1w0$	?	-	$< x; 1w0/?/- >$	CFuw
13	x	$0r0$	0	1	$< x; 0r0/0/1 >$	CFir	14	x	$0r0$	0	?	$< x; 0r0/0/? >$	CFrr
15	x	$0r0$	1	0	$< x; 0r0/1/0 >$	CFdrd	16	x	$0r0$	1	1	$< x; 0r0/1/1 >$	CFrd
17	x	$0r0$	1	?	$< x; 0r0/1/? >$	CFrrd	18	x	$0r0$	?	0	$< x; 0r0/?/0 >$	CFur
19	x	$0r0$	?	1	$< x; 0r0/?/1 >$	CFur	20	x	$0r0$	?	?	$< x; 0r0/?/? >$	CFur
21	x	$1r1$	1	0	$< x; 1r1/1/0 >$	CFir	22	x	$1r1$	1	?	$< x; 1r1/1/? >$	CFrr
23	x	$1r1$	0	0	$< x; 1r1/0/0 >$	CFrd	24	x	$1r1$	0	1	$< x; 1r1/0/1 >$	CFdrd
25	x	$1r1$	0	?	$< x; 1r1/0/? >$	CFrrd	26	x	$1r1$	?	0	$< x; 1r1/?/0 >$	CFur
27	x	$1r1$	?	1	$< x; 1r1/?/1 >$	CFur	28	x	$1r1$	?	?	$< x; 1r1/?/? >$	CFur
29	$0w0$	0	1	-	$< 0w0; 0/1/- >$	CFds	30	$0w0$	0	?	-	$< 0w0; 0/?/- >$	CFud
31	$1w1$	0	1	-	$< 1w1; 0/1/- >$	CFds	32	$1w1$	0	?	-	$< 1w1; 0/?/- >$	CFud
33	$0w1$	0	1	-	$< 0w1; 0/1/- >$	CFds	34	$0w1$	0	?	-	$< 0w1; 0/?/- >$	CFud
35	$1w0$	0	1	-	$< 1w0; 0/1/- >$	CFds	36	$1w0$	0	?	-	$< 1w0; 0/?/- >$	CFud
37	$0r0$	0	1	-	$< 0r0; 0/1/- >$	CFds	38	$0r0$	0	?	-	$< 0r0; 0/?/- >$	CFud
39	$1r1$	0	1	-	$< 1r1; 0/1/- >$	CFds	40	$1r1$	0	?	-	$< 1r1; 0/?/- >$	CFud
41	$0w0$	1	0	-	$< 0w0; 1/0/- >$	CFds	42	$0w0$	1	?	-	$< 0w0; 1/?/- >$	CFud
43	$1w1$	1	0	-	$< 1w1; 1/0/- >$	CFds	44	$1w1$	1	?	-	$< 1w1; 1/?/- >$	CFud
45	$0w1$	1	0	-	$< 0w1; 1/0/- >$	CFds	46	$0w1$	1	?	-	$< 0w1; 1/?/- >$	CFud
47	$1w0$	1	0	-	$< 1w0; 1/0/- >$	CFds	48	$1w0$	1	?	-	$< 1w0; 1/?/- >$	CFud
49	$0r0$	1	0	-	$< 0r0; 1/0/- >$	CFds	50	$0r0$	1	?	-	$< 0r0; 1/?/- >$	CFud
51	$1r1$	1	0	-	$< 1r1; 1/0/- >$	CFds	52	$1r1$	1	?	-	$< 1r1; 1/?/- >$	CFud

The $1PF2_s$ FFMs

Without any operation on a- or v-cells, a certain state of the a-cell sensitizes a fault in the v-cell.

1. **State coupling fault (CFst):** When a certain state in a-cell, reverses the state of v-cell, this is called as a *state coupling fault*. This fault appears due to the initial values of the cells, not by any operation. CFst has 4 FPs: $< 0; 0/1/- >$, $< 0; 1/0/- >$, $< 1; 0/1/- >$ and $< 1; 1/0/- >$.

2. **Undefined State coupling fault (CFus):** When a certain state in a-cell, puts the v-cell into an undefined state, this is called as an *undefined state coupling fault*. CFus has 4 FPs: $< 0; 0/?/- >$, $< 0; 1/?/- >$, $< 1; 0/?/- >$ and $< 1; 1/?/- >$.

The $1PF2_a$ FFMs

An operation on the a-cell sensitizes a fault in v-cell.

1. **Disturb coupling fault (CFds):** When an operation on the a-cell, reverses the v-cell, this is called as a *disturb coupling fault*. CFds has 12 FPs: $< 0w0; 0/\uparrow/- >$, $< 0w0; 1/\downarrow/- >$, $< 1w1; 0/\uparrow/- >$, $< 1w1; 1/\downarrow/- >$, $< 0w1; 0/\uparrow/- >$, $< 0w1; 1/\downarrow/- >$, $< 1w0; 0/\uparrow/- >$, $< 1w0; 1/\downarrow/- >$, $< r0; 0/\uparrow/- >$, $< r0; 1/\downarrow/- >$, $< r1; 0/\uparrow/- >$ and $< r1; 1/\downarrow/- >$.

2. **Undefined Disturb coupling fault (CFud):** When an operation on the a-cell, puts the v-cell into an undefined state, this is called as a *undefined disturb coupling fault.* CFud has 12 FPs: $< 0w0; 0/?/- >$, $< 0w0; 1/?/- >$, $< 1w1; 0/?/- >$, $< 1w1; 1/?/- >$, $< 0w1; 0/?/- >$, $< 0w1; 1/?/- >$, $< 1w0; 0/?/- >$, $< 1w0; 1/?/- >$, $< r0; 0/?/- >$, $< r0; 1/?/- >$, $< r1; 0/?/- >$ and $< r1; 1/?/- >$.

3. **Idempotent coupling fault (CFid):** When a transition write operation ($0w1and1w0$) on the a-cell, sets the v-cell to a non-intended state, 0 or 1 and erroneous value remains on the v-cell, this is called as an *idempotent coupling fault.* CFid has 4 FPs: $< 0w1; 0/\uparrow/- >$, $< 0w1; 1/\downarrow/- >$, $< 1w0; 0/\uparrow/- >$ and $< 1w0; 1/\downarrow/- >$.

4. **Inversion coupling fault (CFin):** When a transition write operation ($0w1$ and $1w0$) on the a-cell, inverts the v-cell whenever it is applied, this is called as an *inversion coupling fault.* CFin has 2 pairs of FPs: $\{< 0w1; 0/\uparrow/- >, < 0w1; 1/\downarrow/- >\}$ and $\{< 1w0; 0/\uparrow/- >, < 1w0; 1/\downarrow/- >\}$. For example, second FP pair means, each $1w0$ operation to the a-cell inverts the v-cell state.

The $1PF2_v$ FFMs

When the a-cell is in a certain state, an operation on the v-cell sensitizes a fault in the v-cell.

1. **Transition coupling fault (CFtr):** When a certain state of the a-cell causes to the failure of a transition write operation on the v-cell, this is called as a *transition coupling fault.* CFtr has 4 FPs: $< 0; 0w1/0/- >$, $< 0; 1w0/1/- >$, $< 1; 0w1/0/- >$ and $< 1; 1w0/1/- >$.

2. **Write Destructive coupling fault (CFwd):** When the a-cell is in a certain state, if a non-transition write operation on the v-cell, causes a transition on v-cell, this is called as a *write destructive coupling fault.* CFwd has 4 FPs: $< 0; 0w0/\uparrow/- >$, $< 0; 1w1/\downarrow/- >$, $< 1; 0w0/\uparrow/- >$ and $< 1; 1w1/\downarrow/- >$.

3. **Read Destructive coupling fault (CFrd):** When the a-cell is in a certain state, if the v-cell is reversed by a read operation on itself and the read-out data is incorrect, this is called as a *read destructive coupling fault.* CFrd has 4 FPs: $< 0; r0/\uparrow/1 >$, $< 0; r1/\downarrow/0 >$, $< 1; r0/\uparrow/1 >$ and $< 1; r1/\downarrow/0 >$.

4. **Deceptive Read Destructive coupling fault (CFdrd):** When the a-cell is in a certain state, if the v-cell is reversed by a read operation on itself, however the read-out data is correct, this is called as a *deceptive read destructive coupling fault.* CFdrd has 4 FPs: $< 0; r0/\uparrow/0 >$, $< 0; r1/\downarrow/1 >$, $< 1; r0/\uparrow/0 >$ and $< 1; r1/\downarrow/1 >$.

5. **Random Read Destructive coupling fault (CFrrd):** When the a-cell is in a certain state, if the v-cell is reversed by a read operation on itself and the read-out data is random, this is called as a *read destructive coupling fault.* CFrrd has 4 FPs: $< 0; r0/\uparrow/? >$, $< 0; r1/\downarrow/? >$, $< 1; r0/\uparrow/? >$ and $< 1; r1/\downarrow/? >$.

6. **Incorrect Read coupling fault (CFir):** When a read operation is performed on the v-cell, with the a-cell is in a certain state, if the read-out data is incorrect, this is called as an *incorrect read coupling fault.* CFir has 4 FPs: $< 0; r0/0/1 >$, $< 0; r1/1/0 >$, $< 1; r0/0/1 >$ and $< 1; r1/1/0 >$.

7. **Random Read coupling fault (CFrr):** When a read operation is performed on the v-cell, with the a-cell is in a certain state, if the read-out data is random, this is called as a *random read coupling fault.* CFrr has 4 FPs: $< 0; r0/0/? >$, $< 0; r1/1/? >$, $< 1; r0/0/? >$ and $< 1; r1/1/? >$.

8. **Undefined Write coupling fault (CFuw):** When a write operation on the v-cell puts the v-cell into an undefined state with a-cell is in a certain state, this is called as an *undefined write coupling fault.* CFuw has 8 FPs: $< x; 0w0/?/- >$, $< x; 1w1/?/- >$, $< x; 0w1/?/- >$ and $< x; 1w0/?/- >$ where $x \in \{0, 1\}$.

9. **Undefined Read coupling fault (CFur):** When a read operation on the v-cell puts the v-cell into an undefined state with a-cell is in a certain state, this is called as an *undefined write coupling fault.* Read-out data can be correct, wrong or random. CFur has 12 FPs: $< x; r0/?/0 >$, $< x; r0/?/1 >$, $< x; r0/?/? >$, $< x; r1/?/0 >$, $< x; r1/?/1 >$ and $< x; r1/?/? >$ where $x \in \{0, 1\}$.

3.3.1.5 Neighborhood pattern sensitive coupling faults

Pattern sensitive fault (PSF) is a generalization of k-coupling faults, where $k = n$ (all cells of memory array). Cell-under-test is called as *base cell. Neighborhood pattern* limits $k = n$ generalization to number of cells surrounding the base cell. Cells remained after the removal of the base cell are named as *deleted neighborhood.* There are three types of NPSFs [100]:

- Static NPSF (SNPSF): A certain pattern of the deleted neighborhood sensitizes the base cell into a certain state.
- Active (Dynamic) NPSF (ANPSF): A certain transition in the pattern of deleted neighborhood sensitizes the base cell.
- Passive NPSF (PNPSF): A certain pattern of the deleted neighborhood prevents the base cell from any change.

3.3.2 Static address decoder faults (sADFs)

To simplify the formulation, it is pre-assumed that an *address decoder fault (AF)* is same during both read and write operations. Secondly, AFs will be derived for a bit-oriented memory where one address accesses to one bit. There are four functional AF fault types as [100]:

1. A particular address can not access to any cell, Figure 3.5 (1),
2. A particular cell can not be accessed by any address, Figure 3.5 (2),

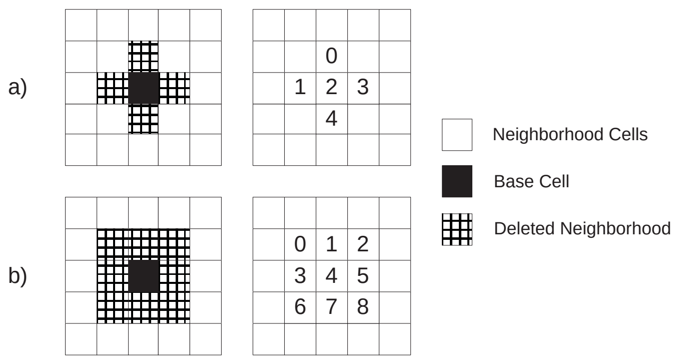


Figure 3.4: a) Type-1 and b) type-2 neighborhood definitions [19]

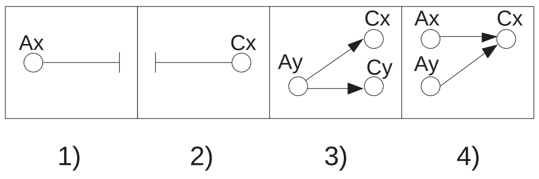


Figure 3.5: Address decoder faults [100]

3. A particular address access to multiple cells, Figure 3.5 (3),
4. A particular cell is accessed by multiple addresses, Figure 3.5 (4),

Obviously, only one of those faults in Figure 3.5 can not exist in a memory cell array, alone. This observation brings us to the *fault combinations*, shown in Figure 3.6 and annotated as fault A to D, respectively [100]. Fault 1 exist with either fault 2 or fault 3; fault 2 with at least one of the fault 1 and 4; fault 3 with at least fault 1 and 4; fault 4 with either fault 2 or 3.

3.3.3 Static peripheral circuit faults (sPCFs)

Those faults occur in the read/write circuits (e.g., write driver, sense amplifier, data register) and precharge circuits. In literature, several proofs are given that read/write logic and address decoder faults can be mapped on the memory cell array faults [100, 80]. This assumption further simplifies previously defined reduced functional memory

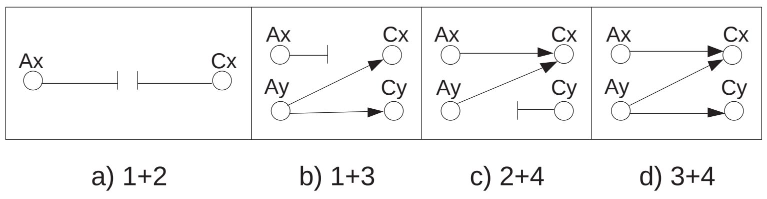


Figure 3.6: Fault combinations of AFs [100]

model in Figure 3.1 to memory cell array. However, then the fault source can not be distinguished anymore; i.e., whether sourced by address decoder, read/write circuit or indeed memory cell array itself.

Low PPM levels mandate to development of dynamic fault primitive-based test algorithms in addition to the static fault tests. Therefore, the following section investigates the dynamic fault space.

3.4 Dynamic faults

Dynamic faults are classified into 3 main classes: memory cell array, address decoder and peripheral circuit faults.

3.4.1 Dynamic memory cell array faults (dMCAFs)

Dynamic faults can be sub-classified as two-operation dynamic FPs ($\#O = 2$), three-operation dynamic FPs ($\#O = 3$), etc. The possibility to trigger a dynamic fault falls down with increasing number of operations [6], thus investigation of dynamic fault space is restricted to the two-operation simple dynamic fault space in literature [46]. A further restriction is made for the number of cells involved, mostly two-cell dynamic faults are considered. To sum up, here, the simple two-operation dynamic faults involving single or two-cells will be explained.

3.4.1.1 Single-cell fault primitives

To generate the whole fault space of two-operation simple dynamic single-cell faults, firstly, all possibilities for each of the S, F and R should be enlisted. Afterwards, they can be easily grouped to generate whole combinations.

To start with S as:

- Let $x, y, z \in \{0, 1\}$. Due to the two-operation restriction, one can have those operation sequences: *write-after-write (xwywz), write-after-read (xwyry), read-after-read (xrxrx)* or *read-after-write (xrxwy)* which resulted in 8, 4, 2 and 4 sensitizing operation sequences, totally 18 Ss, respectively.

for F:

- there are possibly two values, $\{0, 1\}$.

and for R:

- it can take three values, $\{0, 1, -\}$; $\{0, 1\}$ where the last element in sensitizing sequence is a read operation, and $\{-\}$ in case of a write operation.

Those possibilities result in 30 combinations and can be grouped under 5 FFMs as in Table 3.3.

Table 3.3: List of single-cell dynamic FPs and FFMs [46]

FFM	FPs
dRDF	$< 0r0r0/1/1 >$, $< 1r1r1/0/0 >$, $< 0w0r0/1/1 >$, $< 1w1r1/0/0 >$, $< 0w1r1/0/0 >$, $< 1w0r0/1/1 >$
dDRDF	$< 0r0r0/1/0 >$, $< 1r1r1/0/1 >$, $< 0w0r0/1/0 >$, $< 1w1r1/0/1 >$, $< 0w1r1/0/1 >$, $< 1w0r0/1/0 >$
dIRF	$< 0r0r0/0/1 >$, $< 1r1r1/1/0 >$, $< 0w0r0/0/1 >$, $< 1w1r1/1/0 >$, $< 0w1r1/1/0 >$, $< 1w0r0/0/1 >$
dTF	$< 0w0w1/0/- >$, $< 1w1w0/1/- >$, $< 0w1w0/1/- >$, $< 1w0w1/0/- >$, $< 0r0w1/0/- >$, $< 1r1w0/1/- >$,
dWDF	$< 0w0w0/1/- >$, $< 1w1w1/0/- >$, $< 0w1w1/0/- >$, $< 1w0w0/1/- >$, $< 0r0w0/1/- >$, $< 1r1w1/0/- >$,

1. **Dynamic Read Destructive Fault (dRDF):** When the last read operation (read-after-read or write-after-read) in sensitizing sequence disrupts the data in the cell and the returned read data on the output is incorrect, then this fault type is called as a *dynamic read destructive fault (dRDF)*. dRDF has 6 FPs as: $< 0r0r0/1/1 >$, $< 1r1r1/0/0 >$, $< 0w0r0/1/1 >$, $< 0w1r1/0/0 >$, $< 1w0r0/1/1 >$ and $< 1w1r1/0/0 >$.

2. **Dynamic Deceptive Read Destructive Fault (dDRDF):** When the last read operation in sensitizing sequence disrupts the data in the cell, however the returned read data on the output is correct, then this fault type is called as a *dynamic deceptive read destructive fault (dDRDF)*. dDRDF has 6 FPs as: $< 0r0r0/1/0 >$, $< 1r1r1/0/1 >$, $< 0w0r0/1/0 >$, $< 0w1r1/0/1 >$, $< 1w0r0/1/0 >$ and $< 1w1r1/0/1 >$.

3. **Dynamic Incorrect Read Fault (dIRF):** When the last read operation in sensitizing sequence does not disrupt the data in the cell, however the returned read data on the output is incorrect, then this fault type is called as a *dynamic incorrect read fault (dIRF)*. dIRF has 6 FPs as: $< 0r0r0/0/1 >$, $< 1r1r1/1/0 >$, $< 0w0r0/0/1 >$, $< 0w1r1/1/0 >$, $< 1w0r0/0/1 >$ and $< 1w1r1/1/0 >$.

4. **Dynamic Transition Fault (dTF):** When a transition write operation (read-after-write or write-after-write) in sensitizing sequence does not succeed, then this fault type is called as a *dynamic transition fault (dTF)*. dTF has 6 FPs as: $< 0r0w1/0/- >$, $< 1r1w0/1/- >$, $< 0w0w1/0/- >$, $< 1w0w1/0/- >$, $< 0w1w0/1/- >$ and $< 1w1w0/1/- >$.

5. **Dynamic Write Destructive Fault (dWDF):** When a non-transition write operation (read-after-write or write-after-write) in sensitizing sequence disrupts the data in the cell, then this fault type is called as a *dynamic write destructive fault (dWDF)*. dWDF has 6 FPs as: $< 0w0w0/1/- >$, $< 0w1w1/0/- >$, $< 1w0w0/1/- >$, $< 1w1w1/0/- >$, $< 0r0w0/1/- >$ and $< 1r1w1/0/- >$.

Table 3.4: List of two-cell dynamic FPs and FFMs caused by S_{aa} [46]

FFM	FPs
dCFds$_{ww}$	$< 0w0w0; x/\overline{x}/- >$, $< 1w1w1; x/\overline{x}/- >$, $< 0w0w1; x/\overline{x}/- >$, $< 1w1w0; x/\overline{x}/- >$, $< 0w1w0; x/\overline{x}/- >$, $< 1w0w1; x/\overline{x}/- >$, $< 0w1w1; x/\overline{x}/- >$, $< 1w0w0; x/\overline{x}/- >$,
dCFds$_{wr}$	$< 0w0r0; x/\overline{x}/- >$, $< 1w1r1; x/\overline{x}/- >$, $< 0w1r1; x/\overline{x}/- >$, $< 1w0r0; x/\overline{x}/- >$,
dCFds$_{rw}$	$< 0r0w0; x/\overline{x}/- >$, $< 1r1w1; x/\overline{x}/- >$, $< 0r0w1; x/\overline{x}/- >$, $< 1r1w0; x/\overline{x}/- >$,
dCFds$_{rr}$	$< 0r0r0; x/\overline{x}/- >$, $< 1r1r1; x/\overline{x}/- >$,

3.4.1.2 Two-cell fault primitives

As in the two-cell FPs of simple static single-port fault class (see Chapter 3.3.1.3-.4), again, cells are named as the *aggressor a-cell* and the *victim (v-cell)*. In this case, since the number of sequential operations in S is restricted to 2, four possible situations are resulted:

1. S_{aa}: where both of the operations are applied to the a-cell, while v-cell is in a certain state.
2. S_{vv}: where both of the operations are applied to the v-cell, while a-cell is in a certain state.
3. S_{av}: where the first operation is applied to the a-cell and the second one is to the v-cell.
4. S_{va}: where the first operation is applied to the v-cell and the second one is to the a-cell.

Faults caused by $\mathbf{S_{aa}}$

In S_{aa} class faults, two sequential operations applied to the a-cell, resulting in a state change (reverse, flip) of the v-cell. Since both of the two operations will be applied to the a-cell, while the v-cell is in a certain state, $< S_{aa}/F/R >$ will be shown as: $< S_a; S_v/F/R >$.

S_a will be shown as: yO_1zO_2t, where O_1 and O_2 are the first and second operations. They can be w or r operations. S_v will be shown as: x. All $x, y, z, t \in \{0, 1\}$. As a result of state flip, F will be $\overline{x}$. R will be $\{-\}$, since no operations are applied to the v-cell.

In result, $< S_a; S_v/F/R >$ have the general form of $< yO_1zO_2t; x/\overline{x}/- >$. yO_1zO_2t sequence gives 18 possible Ss as in single-cell FPs (see Chapter 3.4.1.1). Thus, there will be a total of 36 FPs grouped under a single FFM as shown in Table 3.4.

- *Dynamic Disturb Coupling Faults (dCFds)*: When both operations are applied to the a-cell while the v-cell in a certain state, results in a state flip of the v-cell. Due to the read/write combinations of operations O_1 and O_2, CFds can be further divided into 4 sub-groups:

1. dCFds$_{ww}$: When both of O_1 and O_2 are a write operation. dCFds$_{ww}$ has 16 FPs as: $< 0w0w0; x/\overline{x}/- >$, $< 0w0w1; x/\overline{x}/- >$, $< 0w1w0; x/\overline{x}/- >$,

Table 3.5: List of two-cell dynamic FPs and FFMs caused by S_{vv} [46]

FFM	FPs
dCFrd	$< x; 0r0r0/1/1 >$, $< x; 1r1r1/0/0 >$, $< x; 0w0r0/1/1 >$, $< x; 1w1r1/0/0 >$, $< x; 0w1r1/0/0 >$, $< x; 1w0r0/1/1 >$
dCFdrd	$< x; 0r0r0/1/0 >$, $< x; 1r1r1/0/1 >$, $< x; 0w0r0/1/0 >$, $< x; 1w1r1/0/1 >$, $< x; 0w1r1/0/1 >$, $< x; 1w0r0/1/0 >$
dCFir	$< x; 0r0r0/0/1 >$, $< x; 1r1r1/1/0 >$, $< x; 0w0r0/0/1 >$, $< x; 1w1r1/1/0 >$, $< x; 0w1r1/1/0 >$, $< x; 1w0r0/0/1 >$
dCFtr	$< x; 0w0w1/0/- >$, $< x; 1w1w0/1/- >$, $< x; 0w1w0/1/- >$, $< x; 1w0w1/0/- >$, $< x; 0r0w1/0/- >$, $< x; 1r1w0/1/- >$,
dCFwd	$< x; 0w0w0/1/- >$, $< x; 1w1w1/0/- >$, $< x; 0w1w1/0/- >$, $< x; 1w0w0/1/- >$, $< x; 0r0w0/1/- >$, $< x; 1r1w1/0/- >$,

$< 0w1w1; x/\overline{x}/- >$, $< 1w0w0; x/\overline{x}/- >$, $< 1w0w1; x/\overline{x}/- >$, $< 1w1w0; x/\overline{x}/- >$ and $< 1w1w1; x/\overline{x}/- >$.

2. dCFds$_{wr}$: When O_1 is a write and O_2 is a read operation. dCFds$_{wr}$ has 8 FPs as: $< 0w0r0; x/\overline{x}/- >$, $< 0w1r1; x/\overline{x}/- >$, $< 1w0r0; x/\overline{x}/- >$ and $< 1w1r1; x/\overline{x}/- >$.

3. dCFds$_{rr}$: When both of O_1 and O_2 are a read operation. dCFds$_{rr}$ has 4 FPs as: $< 0r0r0; x/\overline{x}/- >$ and $< 1r1r1; x/\overline{x}/- >$.

4. dCFds$_{rw}$: When O_1 is a read and O_2 is a write operation. dCFds$_{rw}$ has 8 FPs as: $< 0r0w0; x/\overline{x}/- >$, $< 0r0w1; x/\overline{x}/- >$, $< 1r1w0; x/\overline{x}/- >$ and $< 1r1w1; x/\overline{x}/- >$.

Faults caused by $\mathbf{S_{vv}}$

In S_{vv} class faults, with the a-cell in a certain state, two sequential operations applied to the v-cell triggers a fault in the v-cell. Since both of the two operations will be applied to the v-cell while the a-cell is in a certain state, $< S_{vv}/F/R >$ will be shown as: $< S_a; S_v/F/R >$.

S_a will be shown as: x. S_v will be shown as: yO_1zO_2t, where O_1 and O_2 are the first and second operations. They can be w or r operations. All $x, y, z, t \in \{0, 1\}$. F is the fault appearing in the v-cell and R is the data read-out after a read operation in O_2.

In result, $< S_a; S_v/F/R >$ have the general form of $< x; yO_1zO_2t/F/R >$. There will be a total of 60 FPs grouped under five FFMs as shown in Table 3.5.

1. *Dynamic Read Destructive Coupling Fault (dCFrd)*: is observed while the a-cell in a certain state, if the last operation in the sequence is a read to the v-cell and flips the data in the v-cell and returns an incorrect data at the read-out. dCFrd has the forms of $< x; ywzrz/\overline{z}/\overline{z} >$ and $< x; yryry/\overline{y}/\overline{y} >$ which result in 12 FPs as: $< x; 0w0r0/1/1 >$, $< x; 0w1r1/0/0 >$, $< x; 1w0r0/1/1 >$, $< x; 1w1r1/0/0 >$, $< x; 0r0r0/1/1 >$ and $< x; 1r1r1/0/0 >$.

2. *Dynamic Deceptive Read Destructive Coupling Fault (dCFdrd)*: is observed while the a-cell in a certain state, if the last operation in the sequence is a read to the v-cell and flips the data in the v-cell, however returns a correct data at the read-out. dCFdrd has the forms of $< x; ywzrz/\overline{z}/z >$ and $< x; yryry/\overline{y}/y >$ which result in 12 FPs as: $< x; 0w0r0/1/0 >$, $< x; 0w1r1/0/1 >$, $< x; 1w0r0/1/0 >$, $< x; 1w1r1/0/1 >$, $< x; 0r0r0/1/0 >$ and $< x; 1r1r1/0/1 >$.

3. *Dynamic Incorrect Read Coupling Fault (dCFir)*: is observed while the a-cell in a certain state, if the last operation in the sequence is a read to the v-cell, however does not flip the data in the v-cell, but returns an incorrect data at the read-out. dCFir has the forms of $< x; ywzrz/z/\overline{z} >$ and $< x; yryry/y/\overline{y} >$ which result in 12 FPs as: $< x; 0w0r0/0/1 >$, $< x; 0w1r1/1/0 >$, $< x; 1w0r0/0/1 >$, $< x; 1w1r1/1/0 >$, $< x; 0r0r0/0/1 >$ and $< x; 1r1r1/1/0 >$.

4. *Dynamic Transition Coupling Fault (dCFtr)*: is observed while the a-cell in a certain state, if the last operation in the sequence is a transition write to the v-cell and does not succeed in the write operation. dCFtr has the forms of $< x; ywzw\overline{z}/z/- >$ and $< x; yryw\overline{y}/y/- >$ which result in 12 FPs as: $< x; 0w0w1/0/- >$, $< x; 0w1w0/1/- >$, $< x; 1w0w1/0/- >$, $< x; 1w1w0/1/- >$, $< x; 0r0w1/0/- >$ and $< x; 1r1w0/1/- >$.

5. *Dynamic Write Destructive Coupling Fault (dCFwd)*: is observed while the a-cell in a certain state, if the last operation in the sequence is a non-transition write to the v-cell and flips the data in the v-cell. dCFwd has the forms of $< x; ywzwz/\overline{z}/- >$ and $< x; yrywy/\overline{y}/- >$ which result in 12 FPs as: $< x; 0w0w0/1/- >$, $< x; 0w1w1/0/- >$, $< x; 1w0w0/1/- >$, $< x; 1w1w1/0/- >$, $< x; 0r0w0/1/- >$ and $< x; 1r1w1/0/- >$.

Faults caused by $\mathbf{S_{av}}$

In S_{av} class faults, the first operation applied to the a-cell, afterwards the second operation to the v-cell triggers a fault in the v-cell. $< S_{av}/F/R >$ will be shown as: $< S_a; S_v/F/R >_{av}$.

S_a will be shown as: xO_1y. S_v will be shown as: zO_2t, where O_1 and O_2 are the first and second operations. They can be w or r operations. All $x, y, z, t \in \{0, 1\}$. Thus, S_a and S_v can be one of $\{0w0, 0w1, 1w0, 1w1, 0r0, 1r1\}$ operations. F is the fault appears in the v-cell and R is the data read-out after a read operation in O_2.

- when S_v is zwt, F becomes $\{\overline{t}\}$ and R becomes $\{-\}$. In this case, S_{av} faults have the form of $< S_a; zwt/\overline{t}/- >$ which results in 24 FPs.

- when S_v is zrz,

  – if F is $\{\overline{z}\}$, R becomes one of $\{\overline{z}, z\}$. In this case, S_{av} faults have the form of $< S_a; zrz/\overline{z}/\overline{z} >$and $< S_a; zrz/\overline{z}/z >$ which result in totally 24 FPs.

  – if F is $\{z\}$, R becomes one of $\{\overline{z}\}$. In this case, S_{av} faults have the form of $< S_a; zrz/z/\overline{z} >$ which results in totally 12 FPs.

Table 3.6: List of two-cell dynamic FPs and FFMs caused by S_{av} [46]

FFM	FPs
dCFrd	$< xOy; 0r0/1/1 >$, $< xOy; 1r1/0/0 >$
dCFdrd	$< xOy; 0r0/1/0 >$, $< xOy; 1r1/0/1 >$
dCFir	$< xOy; 0r0/0/1 >$, $< xOy; 1r1/1/0 >$
dCFtr	$< xOy; 0w1/0/- >$, $< xOy; 1w0/1/- >$,
dCFwd	$< xOy; 0w0/1/- >$, $< xOy; 1w1/0/- >$,

There will be a total of 60 FPs grouped under five FFMs as shown in Table 3.6. Names of the FFMs are same as the FFMs of previous fault class caused by S_{vv}.

1. *Dynamic Read Destructive Coupling Fault (dCFrd)*: is observed when the operation O_2 is a read to the v-cell and flips the data in the v-cell and returns an incorrect data at the read-out. dCFrd has the forms of $< S_a; zrz/\overline{z}/\overline{z} >$ which results in 12 FPs as: $< xOy; 0r0/1/1 >$ and $< xOy; 1r1/0/0 >$.

2. *Dynamic Deceptive Read Destructive Coupling Fault (dCFdrd)*: is observed when the operation O_2 is a read to the v-cell and flips the data in the v-cell, however returns a correct data at the read-out. dCFdrd has the forms of $< S_a; zrz/\overline{z}/z >$ which results in 12 FPs as: $< xOy; 0r0/1/0 >$ and $< xOy; 1r1/0/1 >$.

3. *Dynamic Incorrect Read Coupling Fault (dCFir)*: is observed when the operation O_2 is a read to the v-cell, however does not flip the data in the v-cell, but returns an incorrect data at the read-out. dCFir has the forms of $< S_a; zrz/z/\overline{z} >$ which results in 12 FPs as: $< xOy; 0r0/0/1 >$ and $< xOy; 1r1/1/0 >$.

4. *Dynamic Transition Coupling Fault (dCFtr)*: is observed when the operation O_2 is a transition write to the v-cell and does not succeed in the write operation. dCFtr has the forms of $< S_a; zw\overline{z}/z/- >$ which results in 12 FPs as: $< xOy; 0w1/0/- >$ and $< xOy; 1w0/1/- >$.

5. *Dynamic Write Destructive Coupling Fault (dCFwd)*: is observed when the operation O_2 is a non-transition write to the v-cell and flips the data in the v-cell. dCFwd has the forms of $< S_a; zwz/\overline{z}/- >$ which result in 12 FPs as: $< xOy; 0w0/1/- >$ and $< xOy; 1w1/0/- >$.

Faults caused by $\mathbf{S_{va}}$

In S_{va} class faults, the first operation applied to the v-cell, afterwards the second operation to the a-cell triggers a fault in the v-cell. $< S_{va}/F/R >$ will be shown as: $< S_v/F/R; S_a >_{va}$. The number of faults and FFMs are the same as the previous fault class that is caused by S_{av}, only the access order to the a-cell and the v-cell is reverse in this case.

S_a will be shown as: xO_2y. S_v will be shown as: zO_1t, where O_1 and O_2 are the first and second operations. They can be w or r operations. All $x, y, z, t \in \{0, 1\}$. Thus, S_a and S_v can be one of $\{0w0, 0w1, 1w0, 1w1, 0r0, 1r1\}$ operations. F is the fault appears in the v-cell and R is the data read-out after a read operation in O_2.

Table 3.7: List of two-cell dynamic FPs and FFMs caused by S_{va} [46]

FFM	FPs
dCFrd	$< 0r0/1/1; xOy >$, $< 1r1/0/0; xOy >$
dCFdrd	$< 0r0/1/0; xOy >$, $< 1r1/0/1; xOy >$
dCFir	$< 0r0/0/1; xOy >$, $< 1r1/1/0; xOy >$
dCFtr	$< 0w1/0/-; xOy >$, $< 1w0/1/-; xOy >$,
dCFwd	$< 0w0/1/-; xOy >$, $< 1w1/0/-; xOy >$,

- when S_v is zwt, F becomes $\{\overline{t}\}$ and R becomes $\{-\}$. In this case, S_{va} faults have the form of $< zwt/\overline{t}/-; S_a >$ which results in 24 FPs.
- when S_v is zrz,
  - if F is $\{\overline{z}\}$, R becomes one of $\{\overline{z}, z\}$. In this case, S_{va} faults have the form of $< zrz/\overline{z}/\overline{z}; S_a >$and $< zrz/\overline{z}/z; S_a >$ which result in totally 24 FPs.
  - if F is $\{z\}$, R becomes one of $\{\overline{z}\}$. In this case, S_{va} faults have the form of $< zrz/z/\overline{z}; S_a >$ which results in totally 12 FPs.

There will be a total of 60 FPs grouped under five FFMs as shown in Table 3.7. Names of the FFMs are same as the FFMs of previous fault class caused by S_{vv} and S_{av}.

1. *Dynamic Read Destructive Coupling Fault (dCFrd)*: is observed when the operation O_1 is a read to the v-cell and flips the data in the v-cell and returns an incorrect data at the read-out. dCFrd has the forms of $< zrz/\overline{z}/\overline{z}; S_a >$ which results in 12 FPs as: $< 0r0/1/1; xOy >$ and $< 1r1/0/0; xOy >$.

2. *Dynamic Deceptive Read Destructive Coupling Fault (dCFdrd)*: is observed when the operation O_1 is a read to the v-cell and flips the data in the v-cell, however returns a correct data at the read-out. dCFdrd has the forms of $< zrz/\overline{z}/z; S_a >$ which results in 12 FPs as: $< 0r0/1/0; xOy >$ and $< 1r1/0/1; xOy >$.

3. *Dynamic Incorrect Read Coupling Fault (dCFir)*: is observed when the operation O_1 is a read to the v-cell, however does not flip the data in the v-cell, but returns an incorrect data at the read-out. dCFir has the forms of $< zrz/z/\overline{z}; S_a >$ which results in 12 FPs as: $< 0r0/0/1; xOy >$ and $< 1r1/1/0; xOy >$.

4. *Dynamic Transition Coupling Fault (dCFtr)*: is observed when the operation O_1 is a transition write to the v-cell and does not succeed in the write operation. dCFtr has the forms of $< zw\overline{z}/z/-; S_a >$ which results in 12 FPs as: $< 0w1/0/-; xOy >$ and $< 1w0/1/-; xOy >$.

5. *Dynamic Write Destructive Coupling Fault (dCFwd)*: is observed when the operation O_1 is a non-transition write to the v-cell and flips the data in the v-cell. dCFwd has the forms of $< zwz/\overline{z}/-; S_a >$ which result in 12 FPs as: $< 0w0/1/-; xOy >$ and $< 1w1/0/-; xOy >$.

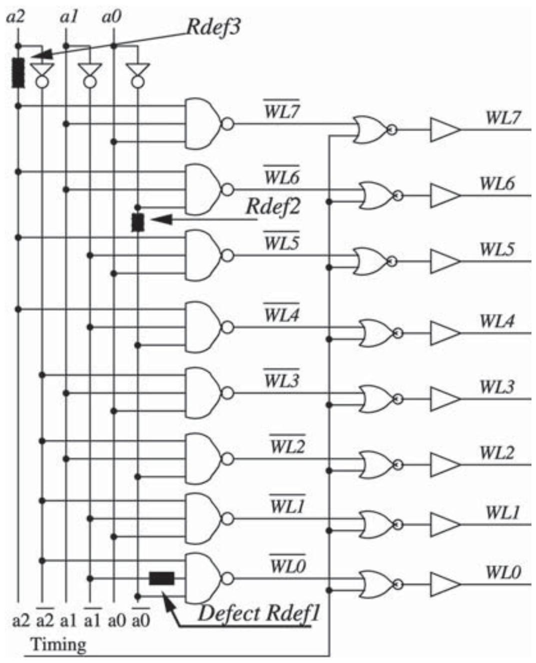


Figure 3.7: A typical CMOS address decoder and example of an intergate open [44]

3.4.2 Dynamic address decoder faults (dADFs)

Address decoder delay faults are mainly caused by the resistive opens on the address decoding logic paths. Resistive opens have two types as [44, 90, 11, 81, 65]:

1. *Intergate opens*: Those are the opens between the logic gates of an address decoder as in Figure 3.7.
2. *Intragate opens*: Those are the opens inside a logic gate as in Figure 3.8 that is used to implement an address decoder.

[65] proposes that the probability of an intergate open is one order of magnitude higher than the existence of an intragate open due to the longer global wire connections instead of short local ones.

The dADFs consists of two faults [44]:

1. *Activation Delay Fault (ActD)*: is a delay-related fault on the rising edge of WL signal due to resistive defects. ActD can be observed both on inter/intragate open situations.
2. *De-activation Delay Fault (DeactD)*: is a delay-related fault on the falling edge of WL signal due to resistive defects and can be observed both on inter/intragate open situations.

Figure 3.9 shows an example of gradually increasing open defects in the row address decoder: the defect value has a direct impact on the delay amount.

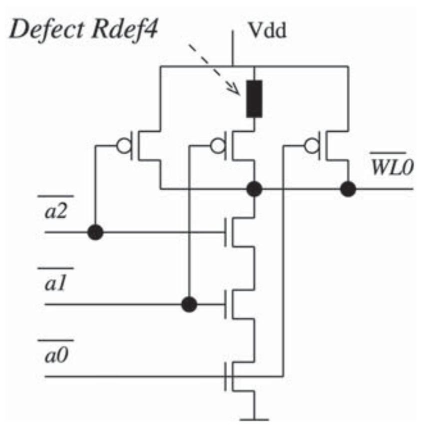


Figure 3.8: A 3-input NAND gate and example of an intragate open [44]

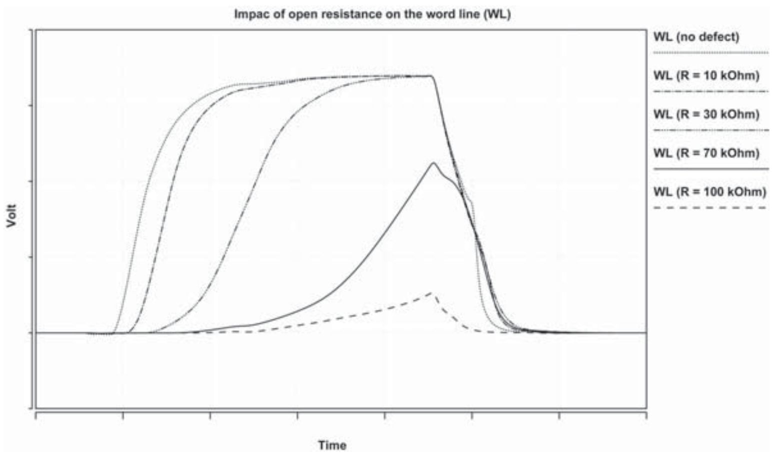


Figure 3.9: Timing diagram for a good and bad WL [44]

3.4.3 Dynamic peripheral circuit faults (dPCFs)

This class of faults occur as a speed-related misbehavior or an excessive leakage on the peripheral circuits. Speed-related faults are presented in the write drivers, sense amplifiers or pre-charge circuit, whereas excessive leakage is related to the pass transistors [107]. dPCFs are divided into four types as:

1. *Slow Write Driver Fault (SWDF)*: Due to a defect in the write driver and/or a resistive defect (i.e., a partial open via) on the path between the write driver and the cell to be written, voltage difference between the BL and $\overline{BL}$ is decreased, resulting in an unsuccessful write operation.

2. *Slow Sense Amplifier Fault (SSAF)*: Due to a slow sense amplifier or an offset voltage caused by a defect in the sense amplifier and/or a resistive defect on the path between the sense amplifier and the cell to be read, an incorrect read-out is

resulted.

3. *Slow PRecharge circuit Fault (SPRF)*: Due to a slow pre-charge circuit or a defect in the pre-charge circuit and/or a resistive defect on the BLs, BLs can not be charged up to the same voltage level, resulting in an incorrect read-out [3].

4. *Bit Line Imbalance Fault (BLIF)*: To correctly detect the data in a cell during a read operation, the voltage difference on the BLs should be higher than a certain threshold level. With the shrinking process technology sizes, leakage current on transistors is increasing. In an extreme case, let assume a column has cells all storing $\overline{x}$ except one has x. When x-storing cell is accessed for a read operation, voltage difference that should be generated on BLs due to the x value can be decreased under to the required threshold level and even if neutralized by the leaking pass transistors of cells storing $\overline{x}$, resulting in an incorrect read data output [74].

To finalize, this chapter introduced the fault primitives (FPs) and functional fault models (FFMs). Single-port, simple (unlinked), static and dynamic, single and two-cell faults that occur in the memory cell array, address decoder and peripheral circuits were discussed. Next chapter continues with the memory tests to detect those faults.

4 Memory testing

This chapter presents the memory test algorithms. Section 4.1 explains the notation of the memory test algorithms. Section 4.2. overviews the ad-hoc memory tests. Section 4.3 continues with the March tests introduced before the development of the fault primitive concept. Section 4.4 discusses the March tests introduced after the development of the fault primitive concept. Section 4.5 concludes with the repetitive tests.

4.1 Notation of march tests

A *march test* is a finite sequence of *march elements (MEs)* which are separated by a ';'. A ME is a finite sequence of operations performed on each cell before proceeding with a next cell. Each operation is separated by a ','. The whole march test algorithm is written between brackets '{...}' while a march element is shown between brackets '(...)'. Access order to cell array can follow one of two *address orders (AOs)*; either $\Uparrow$ ascending or $\Downarrow$ descending. $\Updownarrow$ symbolizes that AO can be chosen freely.

To be more clear, here is the *MATS* [79] march test written in this notation as: $\{\Updownarrow(w0);\Updownarrow(r0,w1);\Updownarrow(r1)\}$. It has 3 MEs $\Updownarrow(w0)$, $\Updownarrow(r0,w1)$ and $\Updownarrow(r1)\}$. $\Updownarrow$ shows AO is free to choose. Algorithm starts with a $w0$ operation to each memory cell. Once all cells are written zero, then the algorithm proceeds with the next march element $\Uparrow(r0,w1)$. Each cell is read for a zero and written with one. Finally, it checks the value of each cell by a $r1$ operation. It is a linear test algorithm with a complexity of $O(4RC)$, because each cell is accessed 4 times.

Some defects can be only detected by certain addressing like fast row (fast X) or fast column (fast Y) access. Neighboring row and column coupling faults can be detected by fast X and fast Y with incrementing or decrementing by 1 while by 2 can detect open faults on the address decoder logic [83].

Fault coverage varies between several march algorithms and also between memory architectures, etc. Mainly, the complexity of march elements determines the quality of a march test.

In case of multi-port memory tests, the notation given above should be extended as [36]:

- Operations are applied to consecutively named ports; i.e., port a, port b,..., port p, in parallel within same or different AOs.
- Port name for each operation is superscripted and operations are separated by colons as: $(r0^a : r0^b : ... : r0^p)$.
- Character *'n'* refers to no operation and character *'-'* refers to any operation that does not conflict with the rest of group.

- $x^p_{r,c}$ refers to that operation x is applied via the port p to the cell on row r and column c.
- $\Uparrow_{i=0}^{n-2}\Uparrow_{a=j+1}^{n-1}$ refers to that an operation cell i travels from 0 to $n-2$; for each cell i, cell j travels from $j+1$ to $n-1$.

Here is an example of the extension of MATS algorithm for 2P memories, 2P-MATS is derived as: $\{\Updownarrow(w0:n);\Updownarrow(r0:r0,w1:n);\Updownarrow(r1:n)\}$.

Before the systematic development of fault modeling, so called *ad-hoc tests* were applied; however after the formalization of fault primitives and fault modeling theory, *fault primitive based* marching tests have become popular.

Before starting, a little remark about memory dimensions and addressing should be noted. In [100], 2^N is given as the number of address locations, where N is the number of address bits. And n is the number of bits, where $n = B.2^N$. For example, a memory can has 2^8 words and each word can have 8 bits, then n will be $8x256$. It is important because a single operation can be performed by a single word access. Thus initialization of memory with a $w0$ can be speed up to 2^N test time instead of $B.2^N$. In [100], single operations are performed with word access and have a test time of 2^N. Thus to convert the test time from word-oriented to bit-oriented, one should replace 2^N with a RC at the test time formulas of algorithms below.

4.2 Ad-hoc test algorithms

SCAN [1], Checkerboard [18], Walking 1/0 [100], GalPat [18] and Butterfly [100] are well-known ad-hoc tests. Walking 1/0, GalPat, Butterfly and Sliding Diagonal are also known as the base cell tests, since they perform ping-pong operations between a base and victim cells. Although their fault coverage (SCAN, Walking 1/0, Butterfly) or practicality (GalPat, Walking 1/0) is not high enough, those traditional tests were commonly applied before the systematic fault primitives were developed. In the case of static simple two-cell fault coverage, none of them exceeds 61% FC barrier. In the case of static simple single-cell faults, they still can not exceed 67% FC, where GalPat has a 83.3% FC. However, still they have the ability to detect some unique faults.

SCAN

$\{\Uparrow(w0);\Uparrow(r0);\Uparrow(w1);\Uparrow(r1)\}$

SCAN is also known as Zero-One [100] or MSCAN (Memory Scan) [1]. As the name refers, firstly all the memory cells are written with 0, read for 0, then written with 1 and read for 1. SCAN has a 4.2^N test time and a complexity of $O(n)$. It has 100% SAF coverage, however total FC is unacceptable for an industrial test flow. It may be applied to detect faulty chips at an early stage. In an extreme case, just a correctly working single cell but not the rest of memory cell array can make the memory pass from the test. SCAN cannot detect whether each cell is correctly addressed or not.

However, with some modifications in addressing order and data background, its fault coverage and covered fault types can be extended including some dynamic faults. For this purpose, H1-Scan and RaW-Scan are proposed at [38].

Checkerboard

$\{\Uparrow(w1_1, w0_2); \Uparrow(r1_1, r0_2);$
$\Uparrow(w0_1, w1_2); \Uparrow(r0_1, r1_2)\}$

Checkerboard [18] divides the memory cells into two groups as black and whites squares of a checkerboard. It is designed to test leakages between neighboring cells, because a cell with 1 is encircled by cells with zero, thus leakage is stressed. Checkerboard has a 4.2^N test time and a complexity of $O(n)$.

GalPat

$\{\Uparrow(w0); \Uparrow_b(w1_b, \Uparrow_{-b}(r0, r1_b), w0_b);$
$\Uparrow(w1); \Uparrow_b(w0_b, \Uparrow_{-b}(r1, r0_b), w1_b)\}$; *(GalPat)*

$\{\Uparrow(w0); \Uparrow_b(w1_b, \Uparrow_{R-b}(r0, r1_b), w0_b);$
$\Uparrow(w1); \Uparrow_b(w0_b, \Uparrow_{R-b}(r1, r0_b), w1_b)\}$; *(GalRow)*

$\{\Uparrow(w0); \Uparrow_b(w1_b, \Box(r0, r1_b), w0_b);$
$\Uparrow(w1); \Uparrow_b(w0_b, \Box(r1, r0_b), w1_b)\}$; *(Gal9R)*

$\{\Uparrow(w0); \Uparrow_b(w1_b, \Box(w0, r1_b), w0_b);$
$\Uparrow(w1); \Uparrow_b(w0_b, \Box(w1, r0_b), w1_b)\}$; *(Gal9W)*

Galloping Pattern [18] is an address-oriented algorithm, all background cells are initialized to complement of the base cell. Then one cell from background except base cell is read, then base cell is read, then one more from the background is read, and base-cell is read again and this goes till to end of address is reached. This sequence is a kind of ping-pong game between the base cell and the rest of the memory. Then all cells are reversed and this sequence is performed again. GalPat has a $2.(2^N + 2.(RC)^2)$ test time and a complexity of $O(n^2)$. Because of the high length of GalPat, it is generally preferred to decrease the complexity by limiting the number of read operations performed on background cells. If read operation is limited to the column that base-cell belongs into, algorithm is named as GalCol [100], if to the row, as GalRow [18], if to the 4 neighbor cells on the directions of south, west, north and east (i.e., $\diamond$), as Gal5R or Gal5W [106], if to the 8 neighbor cells around the base (i.e., $\Box$), as Gal9R or Gal9W [106]. Depending on the test is one of GalCol, GalRow, Gal5R/W, Gal9R/W; the symbol for the address order of operations $M_{1,1}$ and $M_{3,1}$ becomes$\Uparrow_{C-b}, \Uparrow_{R-b}, \diamond, \Box$, respectively. Moreover, between Gal9R and Gal9W, there is a read-write operation difference in the march elements $M_{1,1,0}$ and $M_{3,1,0}$. When compared to previous algorithms, test length of GalPat is still high, thus Sliding Diagonal and Butterfly tests are developed with shorter test time but with lower fault coverages. GalRow and GalCol have $2.(2^N + 2.RC + 2.RC.\sqrt{RC})$ test time with a complexity of $O(n.\sqrt{n})$.

Walking 1/0

(Walking 1/0)
$\{\Uparrow(w0); \Uparrow_b (w1_b, \Uparrow_{-b} (r0), r1_b, w0_b);$
$\Uparrow(w1); \Uparrow_b (w0_b, \Uparrow_{-b} (r1), r0_b, w1_b)\}$

(WalkRow)
$\{\Uparrow(w0); \Uparrow_b (w1_b, \Uparrow_{R-b} (r0), r1_b, w0_b);$
$\Uparrow(w1); \Uparrow_b (w0_b, \Uparrow_{R-b} (r1), r0_b, w1_b)\}$

A walking sequence means that there is a single cell with a complement data respect to the rest of memory at some time point during test. Walking 1/0 [100] has nearly same test element sequence of GalPat with a slight difference. In GalPat, when one cell from the rest is read also base cell is read. In Walking 1/0, all cells are read and lastly the base-cell, thus base cell is read only once. Thus it decreases test time to $2.(2^N + (RC)^2 + 2.RC)$ still with a complexity of $O(n^2)$. To further decrease the complexity, in WalkRow and WalkCol algorithms, read action over all background cells are limited into only the row or column that base-cell belongs to.

Sliding Diagonal

For each diagonal:
$\{\Uparrow(w0_{-d}, w1_d); \Uparrow(r0_{-d}, r1_d); shift_{diagonal}\};$
when all cells become base-cell at diagonal, restart with reverse data sequence
$\{\Uparrow(w1_{-d}, w0_d); \Uparrow(r1_{-d}, r0_d); shift_{diagonal}\};$

In Sliding Diagonal [100] instead of a single base-cell, base-cells located at the memory diagonal are used. Base-cells are written with the complement of background cells. Then all cells are read and diagonal is shifted towards the other two corners. When all cells became base-cell once, data initialization is reversed and sequence is performed again. Sliding diagonal detects all SAFs and TFs and some AFs and some CFs (due to the several base-cells at the same time on diagonal). Due to the diagonal base-cell are spread over to several rows (words), initialization can not be performed by 2^N, but it takes RC operation time. Then sliding diagonal has $6n + 2n.\sqrt{n}$ test time with a complexity of $O(n.\sqrt{n})$.

Butterfly

$\{\Uparrow(w0); \Uparrow(w1_b, \diamond(r0, r1_b), w0_b);$
$\Uparrow(w1); \Uparrow(w0_b, \diamond(r1, r0_b), w1_b)\}$

In Butterfly [100], memory is initialized as in Walking 1/0. Base-cell is written with the complement of background and base cell walks as in Walking 1/0. However, this time, only the background cells at four directions (north, east, south and west) of the base-cell are read. Depending on the implementation choice, distance of background cells that are read on south, east, north and west directions can change; 1, 2, 3, 4, etc. Butterfly has a $2(2^N + (3+k).RC)$ test time with a complexity of $O(n)$. k is the Butterfly maximum distance which is a design specific choice. For example, if k is chosen as 1, 4 neighboring cells; if k is 2, 8 neighboring cells are accessed during the internal loop of Butterfly test. In case of $k = 1$, test time becomes $2(2^N + 7.RC)$.

4.3 March test algorithms

A marching test marches through the memory while data of cells are changed and remained as changed. Then, test further continues with the next victim cell. To be more clear, after the initialization, all cells have the same state. When marching test is at the half of memory, half is at one state while other half is at another state.

Marching tests can be classified into two sub-groups: 1) tests before the fault primitive concept, and 2) tests after the fault primitive concept. This sectioon discusses the March test introduced before the fault primitive concept.

ATS

$\{\Updownarrow(w0_{\prod_1}, w0_{\prod_2}); \Updownarrow(w1_{\prod_0}); \Updownarrow(r0_{\prod_1}); \Updownarrow(w1_{\prod_1}); \Updownarrow(r0_{\prod_2});$
$\Updownarrow(r1_{\prod_0}, r1_{\prod_1}); \Updownarrow(w0_{\prod_0}, r0_{\prod_0}); \Updownarrow(w1_{\prod_2}, r1_{\prod_2})\}$

Algorithmic Test Sequence (ATS) [66] is an optimal algorithm that detects any single SAF and any combination of SA-0 and SA-1 multiple faults. It divides the memory into 3 regions as:

$\prod_0 = \{A_\mu \equiv 0(modulo3)\}$
$\prod_1 = \{A_\mu \equiv 1(modulo3)\}$
$\prod_2 = \{A_\mu \equiv 2(modulo3)\}$, where A_μ is the memory address μ.

then applies the algorithm above. ATS has a 4.2^N test time and a complexity of $O(n)$.

MATS

$\{\Updownarrow(w0); \Updownarrow(r0, w1); \Updownarrow(r1)\}$

The Modified ATS (MATS) [79] is obtained by reordering the steps of ATS algorithm. Simply, M6 is moved to M0; M7 is combined after M4. It detects all SAFs in the memory cell array and Read/Write circuitry. 4.2^N test time and complexity of $O(n)$ of ATS are still valid for MATS. MATS does not deal with 3 distinct memory regions, thus implementation in a program becomes easier. MATS has the same length as SCAN algorithm whereas with a higher fault coverage.

MATS+

$\{\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0)\}$

MATS+ [1] is applied instead of MATS when the memory process technology is not known. M2 in MATS algorithm is modified by adding one extra write operation. M1 guarantees that each cell can be written and read, thus cells are not at SA-0 or SA-1; M1 and M2 guarantee that writing into one cell does not affect a cell in higher or lower address and all over address decoder correctly works. MATS+ is the optimal test for unlinked SAFs with 5.2^N test time and complexity of $O(n)$.

Marching 1/0

$$\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Downarrow (r1, w0, r0); \\ \Updownarrow (w1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1)\}$$

Marching 1/0 [18] is a complete set detecting AFs, SAFs and TFs. Although it is a complete set, it is not an irredundant test, because M3, M4 and M5 are not essential. Only M0 for initialization; M1 for up transition and address decoder faults and M2 for down transition and address decoder faults are enough and necessary to be a complete set. Thus, Marching 1/0 is a redundant test with 14.2^N test time and complexity of $O(n)$.

MATS++

$$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0)\}$$

MATS++ [100] is developed by eliminating the redundant M3, M4, M5 and the last read operation in M1 from Marching 1/0 algorithm. Fault coverage is still same as Marching 1/0 whereas test is optimized. MATS++ has 6.2^N test time and complexity of $O(n)$.

March A

$$\{\Updownarrow (w0); \Uparrow (r0, w1, w0, w1); \Uparrow (r1, w0, w1); \\ \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$$

March A [94] is a complete, irredundant test for linked idempotent coupling faults. It is the shortest test for AFs, SAFs, TFs not linked with CFids, linked CFids, and some CFins linked with CFids. March A has 15.2^N test time and complexity of $O(n)$.

March B

$$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$$

March B [94] is obtained by the modification of M1 in March A test to catch AFs, SAFs, TFs linked with CFins or CFids, CFins and linked CFids. It is a complete and irredundant test, too. March B has 17.2^N test time and complexity of $O(n)$.

March C

$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Updownarrow (r0);$
$\Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$

March C [73] is developed for unlinked inversion, idempotent two-coupling and dynamic two-coupling faults. M3 is a redundant element that makes March C not optimal. March C has 11.2^N test time and complexity of $O(n)$.

March C-

$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$

March C- [100] is obtained by eliminating the redundant element M3 of March C algorithm. It detects unlinked idempotent, inversion, dynamic and state coupling faults. March C- is the simplest test that checks for the unique address faults (AFs) in the address decoder, see Figure 3.5. Since, M1 to M4 perform two operations, it can be called a two-step unique address test. March C- has 10.2^N test time and complexity of $O(n)$.

March A+, A++, C+, C++

(March C+)
$\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0); \Updownarrow (r0)\}$

(March C++)
$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Updownarrow (r0); \Downarrow (r0, w1); \Downarrow (r1, w0);$
$Del; \Uparrow (r0, w1, r1); Del; \Uparrow (r1)\}$

March A+, A++, C+ and C++ [117] are developed based on the March A and C with same modifications to be able to detect read destructive and data retention faults. (+) algorithms (March A+ and C+) have three read operations instead of one as in their original versions to detect disconnected pull-up/down paths in cells. In (++) algorithms (March A++ and C++), two Del elements, $Del; \Uparrow (r_{\overline{d}}, w_d, r_d); Del; \Uparrow (r_d)$, are appended to end of the original algorithms to detect data retention faults.

March G

$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0);$
$Del; \Updownarrow (r0, w1, r1); Del; \Updownarrow (r1, w0, r0)\}\ (Seq.1)$

$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, w1); \Uparrow (r1, w0, r0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, r1, w0);$
$Del; \Updownarrow (r0, w1, r1); Del; \Updownarrow (r1, w0, r0)\}\ (Seq.2)$

March G [99] is designed to detect SOFs and DRFs by extending March B with the elements $Del; \Updownarrow (r0, w1, r1); Del; \Updownarrow (r1, w0, r0)\}$. It has pause elements to catch retention faults. Last two march elements have three step unique address pattern. It covers AFs, TFs and CFs. Test can be made more symmetrical by distributing the two extra read operations of M1 over M2 and M4 (transform from Seq.1 to Seq.2), and can be easier for BIST implementations. March G has $23.2^N + 2.Del$ test time and complexity of $O(n)$.

March X

$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$

March X [100] is designed to detect unlinked inversion coupling faults. X in the name refers to that algorithm has not been published. It detects AFs, SAFs, unlinked inversion CFs and TFs unlinked to CFins. March X has 6.2^N test time and complexity of $O(n)$.

March Y

$\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Downarrow (r1, w0, r0); \Updownarrow (r0)\}$

March Y [100] is another unpublished test algorithm that is designed to detect linked transition and inversion coupling faults. Y in the name refers to the same situation as in the case of March X. M1 and M2 of March X algorithm are extended by adding read operations to the ends. Thus, March Y detects all faults that are detected by March X (AFs, SAFs, CFins, etc.) and moreover TFs linked to CFins. March Y has 8.2^N test time and complexity of $O(n)$.

Algorithm B

$\{\Updownarrow (w0); \Uparrow (r0, w1, w0, w1); \Uparrow (r1, w0, r0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, r1, w0)\}$

Similar to March A and B, Algorithm B [73] is also designed to detect linked faults consisting of any number of simple faults of the same type. Algorithm B has 17.2^N test time and complexity of $O(n)$.

IFA-6, IFA-9, IFA-13

(IFA-6)
$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0, r0)\}$

(IFA-9)
$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); Del; \Updownarrow (r0, w1); Del; \Updownarrow (r1)\}$

(IFA-13)
$\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0); Del; \Updownarrow (r0, w1); Del; \Updownarrow (r1)\}$

The Inductive Fault Analysis (IFA) [72] is a systematic way of fault generation and detection. IFA can be applied to optimize the geometrical design rules in ICs, maximization of wafer yield and evaluation of design quality. By applying this method, in [24, 25], IFA-6, IFA-9 and IFA-13 were presented which targets complete opens and shorts. Actually, IFA-6N is identical to MATS++ test, where IFA-9N is an extension of March C- with delay elements for DRFs, and IFA-13N, the first test for SOFs and DRFs, is identical to extension of MOVI test for DRFs. IFA-9N and 13N are given to test SRAMs with combinatorial and sequential R/W logic, respectively. Both are works on bit and word oriented SRAMs. IFA-6, IFA-9 and IFA-13 have 6.2^N, $12.2^N + 2.Del$ and $16.2^N + 2.Del$ test times and complexity of $O(n)$.

Moving Inversions and PMOVI

$\{\Downarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0)\}$

Moving Inversions [23] were traditional patterns designed as a shorter option to GalPat. The MOVI name is related to written data into the memory: first all is written with 0, then all is inverted to 1 and goes on. It detects AFs, SAFs, TFs and unlinked CFins and most of the unlinked CFids. MOVI was designed as functional and AC parametric tests. AC parametric test part of MOVI tests the access time of memory which is sum of the address decoder and the read logic delays. AC parametric part applies two read operations to two distinct addresses with complemented data. For example, M1 ends with $r\overline{x}$ and proceeds to a new cell with rx. Thus, M1 through M4 can be used for AC parametric test part. In MOVI, after memory initialization, M1 through M4 are applied as the number of bits in address input (N times). Thus MOVI has $2^N + 12.N.2^N$ test time and complexity of $O(n. \log_2(n))$. To remind, for GalPat, two successive read operations with complemented data are performed between base cell and each of the background cells, thus causing a highly complexity as $O(n^2)$.

The Partial MOVI (PMOVI) [23] is a three-step unique address test and can be used for the functional test part of MOVI. A unique address pattern guarantees the addressing of each address uniquely. M1 through M4 consists 3 operations instead of 2. In PMOVI, each cell is read immediately after written which enables the detection of any write destabilization errors; however in March C-, cells are not immediately read after write operation which gives a possibility to those faults to be masked, because cells can be stabilized during the time period to next march element. PMOVI has 13.2^N test time and complexity of $O(n)$.

March U, U-, UD, UD-

(March U)
$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0, w1); \Downarrow (r1, w0)\}$

(March U-)
$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0); \Uparrow (r0, w1); \Downarrow (r1, w0, w1); \Downarrow (r1, w0)\}$

(March UD)
$\{\Updownarrow(w0);\Uparrow(r0,w1,r1,w0);Del;\Uparrow(r0,w1);Del;\Downarrow(r1,w0,r0,w1);\Downarrow(r1,w0)\}$

(March UD-)
$\{\Updownarrow(w0);\Uparrow(r0,w1,r1,w0);Del;\Uparrow(r0,w1);Del;\Downarrow(r1,w0,w1);\Downarrow(r1,w0)\}$

March U [102] was developed to detect all simple (unlinked) faults in a very reasonable test time with respect to the current tests. To be more specific, SAFs, AFs, SOFs, TFs, single or linked with other CFs, and single CFs are detected. Moreover, homogeneous NPSFs where deleted neighborhood cells have the same value are detected, too. March UD [102] is the extended version of March U to detect DRFs by adding two *Del* elements between M1-M2 and M2-M3. For SRAM testing purposes, March U- [102] and March UD- [102] are the simplified versions by removing the $r0$ operation from M3 and M4, respectively. March U, U-, UD and UD- have 13.2^N, 12.2^N, $13.2^N+2.Del$ and $12.2^N+2.Del$ test times, respectively and complexities of $O(n)$.

March LR, LRD and LRDD

(March LR)
$\{\Updownarrow(w0);\Downarrow(r0,w1);\Uparrow(r1,w0,r0,w1);\Uparrow(r1,w0);\Uparrow(r0,w1,r1,w0);\Updownarrow(r0)\}$

March LR [103] is a test for realistic linked faults. It consists of marching (M0, M1, M3 and M5) and walking (M2 and M4) elements. It is a combination of March C-, walking 0 and 1 elements. March LR detects SOFs, SAFs and linked TFs, CFs and linked CFs consisting two simple CFs. March LR is superior to the previous march tests (March A, B, Algorithm B, etc.) which consider linked faults consisting any number of simple faults of same type, while March LR deals realistic linked faults. To note, March LR also detects homogeneous NPSFs, where deleted neighborhood cells have the same value.

(March LRD)
$\{\Updownarrow(w0);\Downarrow(r0,w1);\Uparrow(r1,w0,r0,w1);\Uparrow(r1,w0);\Uparrow(r0,w1,r1,w0);\Updownarrow(r0);$
$Del;\Updownarrow(r0,w1);Del;\Updownarrow(r1)\}$

When this test sequence is extended with the march elements $Del;\Updownarrow(rx,w\overline{x});Del;\Updownarrow(r\overline{x})$ to the end, linked DRFs are able to be detected and this new sequence is named as March LRD [103].

(March LRDD)
$\{\Updownarrow(w0);\Downarrow(r0,w1);\Uparrow(r1,w0,r0,w1);\Uparrow(r1,w0);\Uparrow(r0,w1,r1,w0);\Updownarrow(r0);$
$Del;\Updownarrow(r0,w1,r1);Del;\Updownarrow(r1)\}$

Moreover, another extension is $Del;\Updownarrow(rx,w\overline{x},r\overline{x});Del;\Updownarrow(r\overline{x})$ to catch simple and double DRFs gives March LRDD [103] that detects single and double DRFs linked or unlinked with other faults. March LR, LRD and LRDD has 14.2^N, $17.2^N+2.Del$ and $18.2^N+2.Del$ test times, respectively and complexity of $O(n)$.

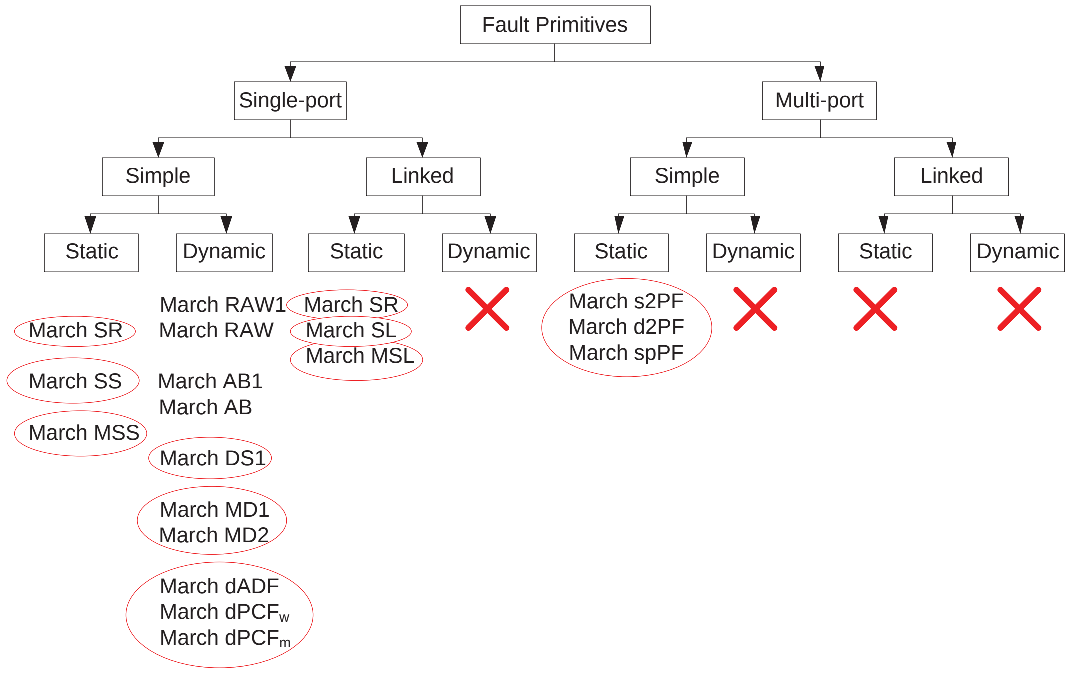


Figure 4.1: Map of the test algorithms to the fault space

March LA, LA-, LAD, LADD-

(March LA)
$\{\Updownarrow(w0); \Uparrow(r0, w1, w0, w1, r1); \Uparrow(r1, w0, w1, w0, r0); \Downarrow(r0, w1, w0, w1, r1);$
$\Downarrow(r1, w0, w1, w0, r0); \Downarrow(r0)\}$

March LA [104] was designed to detect all simple faults as well as all linked faults, consisting any number of simple faults that are known at its design days. However, new fault models have appeared and March LA does not cover all of them, now. Thus, new algorithms have been developed, see March SL. March LA-, LAD and LADD- are the extended versions of March LA to detect DRFs. March LA, March LA-, LAD and LADD- [101] have 22.2^N, 19.2^N, $24.2^N + 2.Del$ and $26.2^N + 2.Del$ test times, respectively and complexity of $O(n)$.

4.4 March tests after the fault primitive concept

Those test developed after the introduction of the fault primitive concept. They are grouped under three sub-classes: static, dynamic and linked fault tests. Figure 4.1 visualizes the classification of test algorithms due to their fault space coverages.

4.4.0.1 Static fault tests

March SS and SR were developed for the static faults. In addition, for multi-port memories, March s2PF, d2PF and spPF tests were developed.

March SR, SR+, SRD and SRD+

$\{\Downarrow(w0); \Uparrow(r0, w1, r1, w0); \Downarrow(r0, r0);$
$\Uparrow(w1); \Downarrow(r1, w0, r0, w1); \Uparrow(r1, r1)\}$ *(March SR)*

$\{\Downarrow(w0); \Uparrow(r0, w1, r1, w0); Del; \Downarrow(r0, r0);$
$\Uparrow(w1); \Downarrow(r1, w0, r0, w1); Del; \Uparrow(r1, r1)\}$ *(March SRD)*

$\{\Downarrow(w0); \Uparrow(r0, r0, w1, r1, r1, w0, r0); \Downarrow(r0);$
$\Uparrow(w1); \Downarrow(r1, r1, w0, r0, r0, w1, r1); \Uparrow(r1)\}$ *(March SR+)*

$\{\Downarrow(w0); \Uparrow(r0, r0, w1, r1, r1, w0, r0); Del; \Downarrow(r0);$
$\Uparrow(w1); \Downarrow(r1, r1, w0, r0, r0, w1, r1); Del; \Uparrow(r1)\}$ *(March SRD+)*

To be able to derive new fault models, the method of resistive defect insertion into the electrical SRAM model was applied as in [109]. Afterwards, observed responses formulated under new FFMs. This method produces FFMs caused by spot defects (SDs) which are classified as open, short or bridge. March SR [48] was designed to detect simple realistic faults. WDF, CFwd and $CFds_{xwx}$ were not considered as realistic faults, therefore they are covered by neither March SR nor SRD. March SRD [48] is the extension of March SR in case of DRFs. March SR+ and SRD+ [37] are reformulated versions of March SR and SRD for better BIST implementations. March SR, SRD, SR+ and SRD+ have 14.2^N, $14.2^N + 2.Del$, 18.2^N and $18.2^N + 2.Del$ test times, respectively and complexity of $O(n)$.

March SS and MSS

$\{\Updownarrow(w0); \Uparrow(r0, r0, w0, r0, w1); \Uparrow(r1, r1, w1, r1, w0);$
$\Downarrow(r0, r0, w0, r0, w1); \Downarrow(r1, r1, w1, r1, w0); \Updownarrow(r0)\}$ *(March SS)*

From the empirical observations on memory tests, existence of additional new FFMs was concluded. Previous tests (i,e. March U, etc.) claiming that all simple static faults were covered, were not enough any more. Thus, March SS [47] was proposed to detect all simple static faults in parallel to recently developed FFMs. It detects all single and two-cell FFMs. March SS has 22.2^N test time and complexity of $O(n)$. Moreover, in case of a BIST implementation, M5 of March SS can be modified as $\Updownarrow(r0, r0, w0, r0, w1)$ to obtain a regular structure.

$\{\Updownarrow(w0); \Uparrow(r0, r0, w1, w1); \Uparrow(r1, r1, w0, w0);$
$\Downarrow(r0, r0, w1, w1); \Downarrow(r1, r1, w0, w0); \Updownarrow(r0)\}$ *(March MSS)*

In [53], several minimal tests (i.e. March M1, M2,..., M7a, M7b,... M7h,..., M14a, etc.) for each simple static FFM class were developed. Last four tests, M14a through M14d, that all consists the same operation sequence in different address order combinations, are named as March MSS1 through MSS4, respectively and they are referred as March MSS. March MSS1 is shown above. March M5 with 9.2^N test time is the minimal test that detects all single-cell faults and test MSS is the minimal one for all single and two-cell faults. March MSS has 18.2^N test time and complexity of $O(n)$.

March s2PF, d2PF and spPF

(March s2PF)

$\{\Updownarrow(w0:n);\Updownarrow(r0:r0,r0:-,w1:r0,r1:r1,r1:-,w0:r1);\Updownarrow(r0:-);$
$\Updownarrow(w1:-);\Updownarrow(r1:r1,r1:-,w0:r1,r0:r0,r0:-,w1:r0);\Updownarrow(r1:-)\}$

(March d2PF)

$\{\Updownarrow(w0:n);\Uparrow_{c=0}^{C-1}(\Uparrow_{r=0}^{R-1}(w1_{r,c}:r0_{r+1,c},w0_{r,c}:r0_{r+1,c}));$
$\Uparrow_{c=0}^{C-1}(\Uparrow_{r=0}^{R-1}(w1_{r,c}:r0_{r,c+1},w0_{r,c}:r0_{r,c+1}));$
$\Updownarrow(w1:-);\Uparrow_{c=0}^{C-1}(\Uparrow_{r=0}^{R-1}(w0_{r,c}:r1_{r+1,c},w1_{r,c}:r1_{r+1,c}));$
$\Uparrow_{c=0}^{C-1}(\Uparrow_{r=0}^{R-1}(w0_{r,c}:r1_{r,c+1},w1_{r,c}:r1_{r,c+1}));$

(March spPF)

$\{\Updownarrow(w0^a:n^b:...:n^p);$
$\Updownarrow(r0^a:r0^b:...:r0^p,r0^a:-^b:...:-^p,w1^a:r0^b:...:r0^p,$
$r1^a:r1^b:...:r1^p,r1^a:-^b:...:-^p,w0^a:r1^b:...:r1^p);$
$\Updownarrow(r0^a:-^b:...:-^p);\Updownarrow(w1^a:-^b:...:-^p);$
$\Updownarrow(r1^a:r1^b:...:r1^p,r1^a:-^b:...:-^p,w0^a:r1^b:...:r1^p,$
$r0^a:r0^b:...:r0^p,r0^a:-^b:...:-^p,w1^a:r0^b:...:r0^p);$
$\Updownarrow(r1^a:-^b:...:-^p)\}$

In case of tests for 2P memories, there are two addressing mechanisms: single-addressing or double-addressing. In single-addressing, only one cell is accessed at a certain time, in other words, two of the ports address the same cell. In double-addressing, each port accesses to a different cell at a certain time. 2P tests are classified due to their addressing mechanisms. In [37], March s2PF and March d2PF were presented, where s and d refer to single and double addressing mechanisms. March s2PF is the test that combines three single addressing tests: March 2PF1, $2PF2_as$ and $2PF2_vs$. The double-addressing test, March $2PF2_{av}$ is referred as March d2PF. Moreover, optimized versions of those tests were developed: March s2PF- and March d2PF- [37] (March d2PF has two optimal versions). March s2PF, d2PF, s2PF- and d2PF- have 16.2^N, 10.2^N, 14.2^N and 9.2^N test times with a complexity of $O(n)$. To remark, those test cover static simple multi-port fault space.

Furthermore, March spPF [37] was developed for single-addressing p-port memories by combining March pPF1, $pPF2_a$ and $pPF2_v$ tests. Also March spPF was optimized to March spPF- [37]. March spPF and spPF- have 16.2^N and 14.2^N test times with a complexity of $O(n)$.

Industrial test results based on Intel caches and STMicroelectronics SRAMs showed the requirement and necessity for new fault types [50, 39]. To catch low DPM levels in the emerging technologies, research was focused on exploring the space of dynamic faults [46] and the design of new test sets.

4.4.0.2 Dynamic fault tests

Those tests were developed for the dynamic faults. In addition, March dADF, $\mathrm{dPCF_w}$ and $\mathrm{dPCF_m}$ were developed for the address decoder and peripheral circuit dynamic faults.

March RAW1, RAW

(March RAW1)
$\{\Updownarrow(w0);\Updownarrow(w0,r0);\Updownarrow(r0);\Updownarrow(w1,r1);\Updownarrow(r1);\Updownarrow(w1,r1);\Updownarrow(r1);\Updownarrow(w0,r0);\Updownarrow(r0)\}$

(March RAW)
$\{\Updownarrow(w0);\Updownarrow(r0,w0,r0,r0,w1,r1);\Updownarrow(r1,w1,r1,r1,w0,r0);\Updownarrow(r0,w0,r0,r0,w1,r1);\Updownarrow(r1,w1,r1,r1,w0,r0);\Updownarrow(r0)\}$

In [42, 51], March RAW1 and March RAW ('read-after-write') tests for single-port, simple, dynamic faults were proposed. All of the single-cell dynamic faults introduced in [46, 42, 51] are covered by March RAW1, while both all single and two-cell dynamic faults are covered by March RAW. To detect only single-cell dynamic faults, March RAW1 can be applied. March RAW1 and RAW have 13.2^N and 26.2^N test times, respectively and complexity of $O(n)$.

March AB1 and AB

(March AB1)
$\{\Updownarrow(w0);\Updownarrow(w1,r1,w1,r1,r1);\Updownarrow(w0,r0,w0,r0,r0)\}$

(March AB)
$\{\Updownarrow(w1);\Downarrow(r1,w0,r0,w0,r0);\Downarrow(r0,w1,r1,w1,r1);\Uparrow(r1,w0,r0,w0,r0);\Uparrow(r0,w1,r1,w1,r1);\Updownarrow(r1)\}$

March AB1 and AB [14] are the optimized versions of March RAW1 and RAW with the same FC. To remark, the complexity of March SS that covers all simple static faults, is same with March AB. Moreover, March AB covers the same set of single-cell dynamic faults as March RAW, as well. Due to its symmetric sequence and ability to detect all simple static and dynamic faults (that proposed in [42, 51]) within the same test complexity, March AB is very attractive for BIST implementations. March AB1 and AB have 11.2^N and 22.2^N test times, respectively and complexity of $O(n)$.

March DS1

$$\{\Updownarrow(w0);\Updownarrow(w0,w0,r0,r0,r0,w0,r0);\Updownarrow(w1,r1,r1,w0,w0,r0,w1,r1);$$
$$\Updownarrow(w1,w1,r1,r1,r1,w1,r1);\Updownarrow(w0,r0,r0,w1,w1,r1,w0,r0);$$
$$\Updownarrow(w0,w1,r1,w0,w1,r1);\Updownarrow(w1,w0,r0,w1,w0,r0)\}$$

The set of single-cell dynamic FFMs (dRDF, dIRF, dDRDF) presented in [42] that used to develop March RAW, have been enlarged to a complete set of two-operation single-cell dynamic faults (dRDF, dIRF, dDRDF, dTF, dWDF) as in [40]. Then, for single-cell dynamic faults (dRDF, DRDF, TF and WDF), tests that are capable of diagnosing during DPM screening (i.e. dRDF-Diag, dTF-Diag, etc.) and optimum versions of those tests in terms of test length (i.e. dRDF-Opt, dTF-Opt, etc.) were proposed in [40, 39]. March DS1 [40] is the one that covers all of those designed tests. It detects all developed dynamic single-cell FFMs, but does not cover two-cells dynamic FFMs. March DS1 has 43.2^N test time and complexity of $O(n)$.

March MD1, MD2

(March MD1b)

$$\{\Updownarrow(w0);\Updownarrow(w0,w1,w0,w1,r1);\Updownarrow(w0,w0);\Updownarrow(w0,w0);\Updownarrow(r0,w1,r1,w1,r1,r1);\Updownarrow(r1);$$
$$\Updownarrow(w1,w0,w1,w0,r0);\Updownarrow(w1,w1);\Updownarrow(w1,w1);\Updownarrow(r1,w0,r0,w0,r0,r0);\Updownarrow(r0)\}$$

(March MD2)

$$\{\Updownarrow(w0);\Uparrow(r0,w1,w1,r1,w1,w1,r1,w0,w0,r0,w0,w0,r0,w0,w1,w0,w1);$$
$$\Uparrow(r1,w0,w0,r0,w0,w0,r0,w1,w1,r1,w1,w1,r1,w1,w0,w1,w0);$$
$$\Downarrow(r0,w1,r1,w1,r1,r1,r1,w0,r0,w0,r0,r0,r0,w0,w1,w0,w1);$$
$$\Downarrow(r1,w0,r0,w0,r0,r0,r0,w1,r1,w1,r1,r1,r1,w1,w0,w1,w0);\Updownarrow(r0)\}$$

In [54], two minimal tests for single and two-cell (of S_{aa} and S_{vv} types) two-operation simple dynamic faults was proposed, March MD1 and MD2, respectively. The single-cell minimal test have two versions, March MD1a and MD1b. When the last read operation in M1 and M6 of March MD1b are moved to head of M2 and M7, March MD1a is obtained. March MD2 is optimized version of March 100N [13] with same fault coverage. It detects all realistic static simple/linked single-port faults as well as dynamic faults. March MD1 and MD2 have 33.2^N and 70.2^N test times and complexity of $O(n)$.

March dADF, dPCF$_\mathbf{w}$ and dPCF$_\mathbf{m}$

(March dADF)

$$\{\Updownarrow(w0);\Uparrow^{H1}(r0,w1);\Uparrow^{H1}(r1,w0);\Downarrow^{H1}(r0,w1);\Downarrow^{H1}(r1,w0)\}$$

(March $dPCF_w$)

$$\{\Updownarrow(w0);x\Updownarrow(w1,r1,w0);\Updownarrow(w1);x\Updownarrow(w0,r0,w1)\}$$

(March $dPCF_m$)

$\{\Updownarrow (w0); x \Updownarrow (r0, w1); x \Updownarrow (r1, w0)\}$

Several tests for address decoder and peripheral circuit faults were proposed. March dADF [44] was developed to detect all dynamic address decoder delay faults (dADFs). $H1$ refers to H1 addressing method. In [107], March $dPCF_w$ (also known as BLIF, March BLI or March BLIWDw that detects SWDF and BLIF faults) and $dPCF_m$ (also known as March SAPRm which detects SSAF and SPRF faults) was presented to target dynamic peripheral circuit faults. x refers to fx address direction. March $dPCF_w$ and $dPCF_m$ should be applied together to cover all PCFs. They are explained in [41]. March dADF, $dPCF_w$ and $dPCF_m$ have $\frac{9}{2}n + 9.n.\log_2(n)$, $8.n$ and $5.n$ test times with a complexity of $O(n.\log_2(n))$, $O(n)$ and $O(n)$, respectively.

4.4.0.3 Linked fault tests

Those tests were developed for the linked faults.

March SL and MSL

(March SL)
$\{\Updownarrow (w0); \Uparrow (r0, r0, w1, w1, r1, r1, w0, w0, r0, w1); \Uparrow (r1, r1, w0, w0, r0, r0, w1, w1, r1, w0);$
$\Downarrow (r0, r0, w1, w1, r1, r1, w0, w0, r0, w1); \Downarrow (r1, r1, w0, w0, r0, r0, w1, w1, r1, w0)\}$

Since March LA, new fault models have appeared. For this purpose, in [43], linked fault space was investigated in details and classified as linked faults involving a single (LF1s), two (LF2s: $LF2_{aa}$, $LF2_{av}$, $LF2_{va}$,) and three cells (LF3s). For each (sub)class, a march algorithm (March LF1, $LF2_{aa}$, etc.) was developed. When those distinct tests are evaluated, it was seen that March $LF2_{aa}$ covers all other faults of (sub)classes and named as March SL [43]. It detects all static linked faults in the (sub)classes of LF1s, $LF2_{aa}$, $LF2_{av}$, $LF2_{va}$ and LF3s. March SL has 41.2^N test time and complexity of $O(n)$. Detailed information on LFs, LF space, developing test algorithms are also presented in [45], briefly.

$\{\Updownarrow (w0); \Uparrow (r0, w1, w1, r1, r1, w0); \Uparrow (r0, w0); \Uparrow (r0); \Uparrow (r0, w1);$
$\Uparrow (r1, w0, w0, r0, r0, w1); \Uparrow (r1, w1); \Uparrow (r1); \Downarrow (r1, w0)\}$ *(March MSL)*

In [55], LF space of [43] was enlarged with new fault primitives by adding 2-composite static faults as the set of all ordered pairs of static fault primitives FP1 and FP2 with the restriction that these FPs can not be sensitized simultaneously. Then, March MSL [55] was developed as seen above. It is a minimal test for detection of all 2-composite static faults including unlinked and realistic linked static faults. March MSL has 23.2^N test time and complexity of $O(n)$.

4.5 Repetitive tests

Hammer tests perform the same operation on a certain base cell several times.

Hammer

(HamWh)
$\{\Uparrow(w0);\Uparrow(r0,w1^h,r1);\Uparrow(r1,w0^h,r0);\Uparrow(r0,w1^h,r1);\Uparrow(r1,w0^h,r0)\}$

(HamRh)
$\{\Uparrow(w0);\Uparrow(r0,w1,r1^h,r1);\Uparrow(r1,w0,r0^h,r0);\Uparrow(r0,w1,r1^h,r1);\Uparrow(r1,w0,r0^h,r0)\}$

(HamWDhrc)
$\{\Uparrow(w0);\nearrow(w1_b^h,\Uparrow_{R-b}(r0),r1_b,\Uparrow_{C-b}(r0),r1_b,w0_b);$
$\Uparrow(w1);\nearrow(w0_b^h,\Uparrow_{R-b}(r1),r0_b,\Uparrow_{C-b}(r1),r0_b,w1_b)\}$

(HamWDhc)
$\{\Uparrow(w0);\nearrow(w1_b^h,\Uparrow_{C-b}(r0),w0_b);$
$\Uparrow(w1);\nearrow(w0_b^h,\Uparrow_{C-b}(r1),w1_b);$

In Hammer tests [106, 98], read/write operations are applied onto the base cell multiple times to provoke partial faults to be full faults. Hammer operations on a base cell are shown as rx^h or wx^h, where h denotes to the number of hammer operations and $\nearrow$ is used in case of increasing address order on main diagonal. HamRh, HamWDhrc and HamWDhc are referred to HamRd, Hammer and HamWr with h values of 16, 1000 and 16, respectively [98]. HamWh, has $(9+4.h).n$, $(13+4.h).n$, $2(n+\sqrt{2n}(h+R+C+1))$ and $(2(n+\sqrt{2n}(h+C)))$ test times with a complexity of $O(hn)$, $O(hn)$ for the first two and, $O(\sqrt{2n}.max(C,h))$ and $O(\sqrt{2n}.max(h,R,C))$.

Still, fault spaces, FFMs and appropriate test developments for single-port linked dynamic; multi-port simple dynamic; and multi-port linked (both static and dynamic) fault classes are waiting to be researched.

5 Memory Built-In Self-Test

As a result of shrinking process technology sizes, memories become more and more embedded into the systems. This has resulted in the complication of the controllability of the inputs and observability of the outputs of an embedded memory. Moreover, design for direct access to deeply embedded memories influences area, placement, routing and design time of chips. In such conditions, design for testability (DFT) has a significant importance. This chapter focuses on Memory BIST methods.

This chapter is organized as follows. Section 5.1 lists the advantages and drawbacks of the Memory BIST compared to DFT. Section 5.2 introduces the types of self-testing. Then, Section 5.3 presents the architecture of Memory BIST. Section 5.4 explains the test architecture. Later on, Section 5.5 classifies the implementation types of Memory BIST. Finally, Section 5.6 concludes by listing the highly appreciated Memory BIST features.

5.1 DFT and BIST aspects

Design For Testability (DFT) is simply the addition of extra circuitry to simplify the observability and controllability problems at internal nodes of a design [100, 34]. In testing, when the test signal generation, and the capture and the analysis of the outputs are partially or fully embedded into the chip, it is called as *Built-In Self-Test (BIST)* as in Figure 5.1. Generally, in DFT, parts in a test with high computational complexity, are embedded into the chip. Thus, DFT concept acts as a kind of hardware accelerator where some parts of the algorithm are still applied externally. While DFT decreases the test time one order of magnitude, BIST decreases 2 to 3 orders of magnitude compared to a conventional test. On the other hand, since the test algorithm is implemented in hardware, area overhead of BIST is a factor of 2 times larger than DFT's [100].

Compared to DFT, Memory BIST is highly favorable due to its advantages as below [100, 4, 52]:

Advantages of Memory BIST: Memory BIST is a **low cost solution**, since it decreases the dependency on the expensive ATE, it does not require any maintenance as in case of ATE. It simplifies and enables the testing of the highly embedded memories which do not have any I/O pin access externally. It is reusable since it is a part of the hardware. Memory BIST is capable of testing any type (e.g., SRAM, DRAM, two-port, three-port, ..., n-port memory) and any number of memories simultaneously. Moreover, it shortens the time-to-market by simplifying the test development procedure. Memory BIST also shortens the overall test time, since the tightly coupled wires speed up the communication to the memory. Furthermore, Memory BIST provides a **high quality**

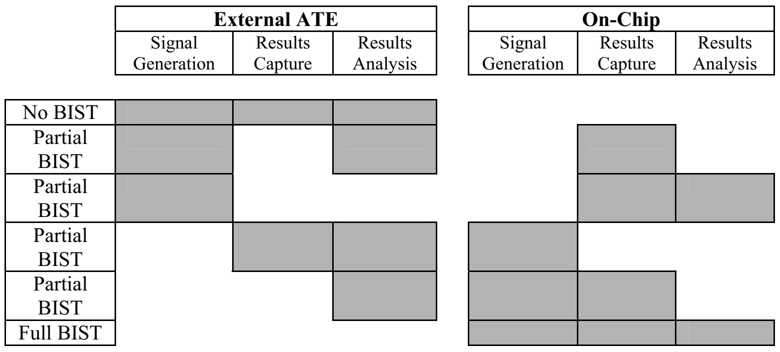


Figure 5.1: No BIST vs partial BIST vs full BIST [34]

memory test with high fault coverage. It is capable of at-speed testing, since it operates at the memory frequency. In addition, there are no intermediate connections (i.e., ATE test header, I/O pins) between the Device-under-Test (DUT) and ATE. Moreover, on-the-fly field testing can be performed whereby faults that only occur in a real-time operation are detected. Finally, it further enables the BISR concept when combined with diagnosis and repair ability.

Drawbacks of Memory BIST: However, there are several drawbacks, too. Memory BIST has an **additional cost**, since it occupies a certain amount of the chip area, and requires extra I/O pins to be controlled and configured. Furthermore, Memory BIST has **to be self-tested**, since it is a hardware. In some cases (e.g., multiplexing), it may degrade the performance. In addition, Memory BIST requires **extra design engineers** specialized in memory testing and BIST implementations.

To sum up, the Memory BIST concept makes possible the testing of highly embedded memories in the system where access to the internal I/O pins of a memory is nearly impossible, costly, unpractical, degree of observability and controllability are too hard.

5.2 Types of Self-Testing

A memory testing procedure can be performed in 3 ways due to the interruption of memory operation and change of memory content at the end of test as [100, 19]:

1. **Concurrent testing:** where the memory is tested simultaneously during its normal operation time. This mechanism has the advantage to detect and correct on-fly faults. Since, memory is in its normal operation, data inside the memory should be preserved and extra error-correcting logic is required. As a result, this feature brings more hardware overhead to BIST engine. Moreover, due to the read/write operations on the cells during normal usage, testing can be interrupted.

2. **Non-concurrent testing:** where the memory is tested in a special test mode. Thus, BIST algorithm can process faster and more freely without any interrupts to memory from system. However, once the testing is finished, content of the memory is lost.

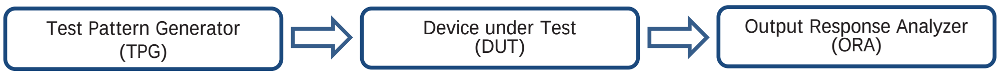


Figure 5.2: General Architecture of the Memory BIST

3. **Transparent testing:** where the memory is interrupted during its normal operation time. Once, the test is finished, content of the memory is loaded back. Thus, this method also requires extra area for the storage of memory content.

In addition, memory testing has two natures depending on the pattern generator used: *pseudo-random (R)* or *deterministic (D)* [100, 2, 52]. This is determined by the three sub-modules of TPG (will be mentioned in next Section 5.3.1): *address (A), control (read or write) (W) and data (D).*

1. **Pseudo-random testing (PR):** is the case when at least one of the test signals of TPG is generated pseudo-randomly as $xAyWzD$, where x, y and $z \in \{R, D\}$ except the case xyz is DDD.

2. **Deterministic testing (DT):** is the case when all test signals of TPG are generated deterministically as $DADWDD$.

5.3 General Architecture of Memory BIST

A Memory BIST mainly consists of three parts: *Test Pattern Generator (TPG), Device-under-Test (DUT), and Output Response Analyzer (ORA)* as in Figure 5.2. TPG, also known as *Test Data Generator (TDG)*, generates the test address, data background and control signals. Those signals are applied to the DUT. DUT is also referred as Circuit-under-Test (CUT), is the memory to-be tested. Lastly, ORA, also known as *Response Data Evaluator (RDE) or Output Data Evaluator (ODE)*, is an output data analyzer. It receives the memory read data, and determines the chip is faulty or not. Here, each will be explained in detail.

5.3.1 Test Pattern Generator (TPG)

TPG consists of three sub-blocks: address, data and control generators. In case of pseudo-random address generator, *linear feedback shift registers (LFSRs)* are applied to generate pseudo-random address and data sequences [100]. An LFSR is basically a shift register. Certain bits in the sequence are led (tapped) to XOR or XNOR gates and the output of those gates are used to re-feed to the shift register sequence [100]. A characteristic polynomial can be written to describe the tap positions. When, the characteristic polynomial is in minimal type, it is called as a primitive polynomial and maximum-length output sequence from LFSR can be achieved.

Figure 5.3 a) shows the Galois (internal or distributed feedback) style, b) shows the Fibonacci (external or centralized feedback) style LFSR implementations with the characteristic polynomial $x^3 + x + 1$. When XOR gates are used, all-zero pattern, when

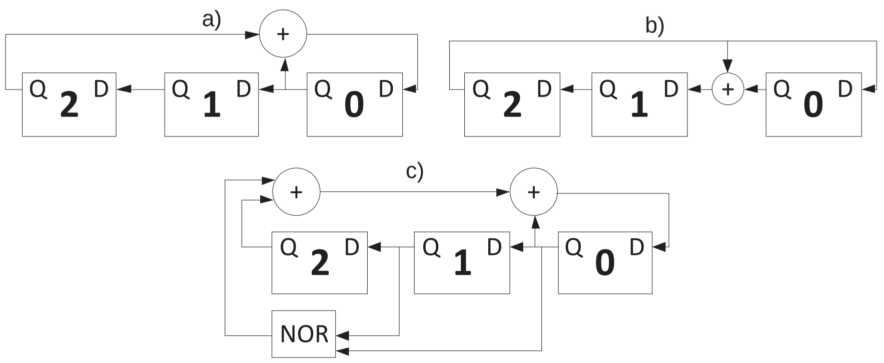


Figure 5.3: LFSR with the characteristic polynomial x^3+x+1 in a) Galois, b) Fibonacci, and c) Galois all-zero pattern styles [100]

Table 5.1: Area comparison of gray-code and binary counters [32]

Address Generator			
Width	Gray-code.(μm^2)	Binary(μm^2)	Difference
8	1350	1853	37.26 %
12	2106	2772	37.26 %
16	2781	3635	30.71 %
20	3448	4614	31.62 %
24	4098	5899	33.82 %

XNOR gates are used, all-one pattern can not be generated. To be more clear, once all of the registers are locked into all-zero states in XOR implementation, LFSR can not proceed into a newer state. This problem can be overcame with one extra NOR and XOR gates as shown in Figure 5.3 c). To obtain the full sequence in PR testing, maximum-length LFSRs are used [100].

In case of deterministic testing, counter (e.g., binary, gray counters) or microprocessor (μP) based address generation can be adapted to TPG. However, those two are not area efficient as LFSR type implementation. On the other hand, memories are highly regular structures where memory tests require accessing to certain addresses in an order as opposed to logic testing [100].

For example, in Table 5.1, an area comparison of gray-code and binary counters is given. Gray-code counter has around 30 to 40% area advantage [32]. Moreover, gray-code counter has a lower power dissipation due to its lower switching activity (single-bit switching) compared to the binary counter. For N-bit counters, binary counter experiences $2^{N+1} - N - 2$ transitions whereas gray-code does 2^N transitions, which is roughly half of the binary [21].

For the data background generation, a cyclic shift register (CSR) or finite-state machine (FSM) might be adopted. For CSR, [112] examined 40 industrial march test algorithms, and combined 252 March elements of those algorithms into 7 *dominant steps.* A dominant step is capable of generating several March elements; for example, r, rw and rwr March elements can be easily generated by the dominant step rwr. Other option

Table 5.2: Area comparison of FSM and CSR based data background generators [32]

Data Background Pattern Generator			
Width	FSM(μm^2)	CSR(μm^2)	Difference
8	382	1273	233.25 %
16	793	2395	202.02 %
32	1102	4622	319.42 %

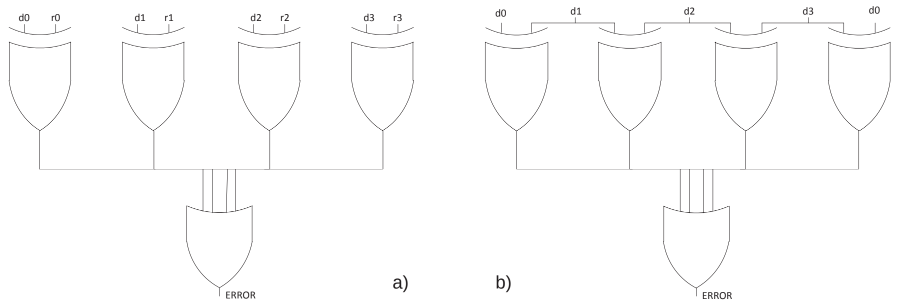


Figure 5.4: a) Direct and b) mutual comparison

is the FSM based pattern generation, where for a memory with N-bit word width, all data background patterns can be realized in $\log_2 N + 1$ states. Table 5.2 shows the area comparison of CSR and FSM based implementations [32]. FSM based data background generation has an area of one third-fourth of CSR based implementation. Moreover, CSR based design requires extra multiplexing and control logic for state transitions. Thus, FSM is an area efficient and faster solution compared to the CSR.

5.3.2 Output Response Analyser (ORA)

For the ORA, several methods are available. Output response from the DUT can be directly or mutually compared with a reference (expected) data; output response can be compressed; or output response can be compacted [4].

The main difference between compression and compaction is that the compression is a reversible process. Meaning that the response data of DUT can be obtained back by decompressing the result of compression, hence it has a lossless characteristic. Whereas, in compaction, the reverse process from the result of compaction to output response of DUT may not be possible. Thus the compaction is lossy in its nature. For example, a faulty and non-faulty DUTs produce different output responses, however final results from the compaction may be same for both; and this situation is called as *aliasing*. On the other hand, compressing huge data, storing the result on-chip, and in case of diagnosis, decompressing this data is not appreciated and not practical for area limited VLSI systems. Therefore, comparator and compaction based ORAs will be overviewed here.

First option for ORA is using a comparator. Figure 5.4 shows deterministic and mutual comparisons [100]. In *deterministic comparison*, generated test data in TPG module, also referred as the *reference data*, is compared with the data readout by using a XOR (unequality checker) gate. In case of multiple memory cell arrays, results of

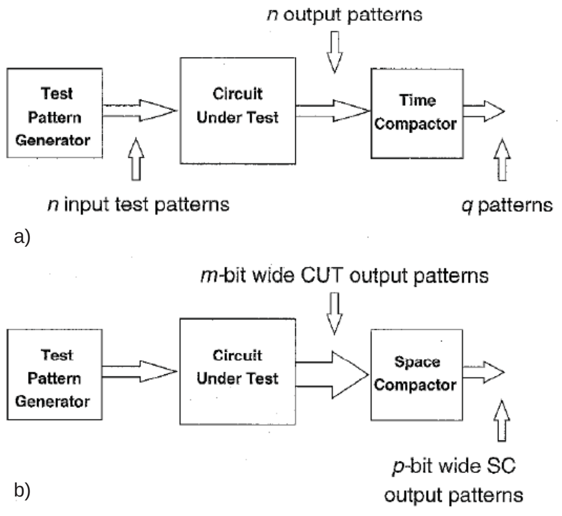


Figure 5.5: a) Time and b) space compaction [59]

XOR gates are led to an OR gate as shown in Figure 5.4 a). Output of the OR gate is triggered to logic-1 in case of any mismatch between reference and readout data. However, for a direct comparison, the reference data generated by TPG, which is closely located to the inputs of memory, should be transfered to the ORA, which is located near to the output of memory. This situation may result in placement and routing difficulties. As shown in Figure 5.4, *mutual comparison* compares the memory data with each other, instead of comparing readout data with a reference. Any mismatch between them means that a faulty situation does exist. While direct comparators can be used in deterministic not in pseudo-random testing, mutual comparators can be used both for deterministic and pseudo-random testings [100]. Advantage of a comparator based ORA is the localization of fault at the time of its occurrence.

A second option for ORA is the compaction. Compaction can be done in two domains: time or space. In *time compaction (TC)*, a longer bit-stream in the range of 10^{5-6} is compacted into a shorter bit-stream that consists of 16 to 32 bits [59] after a certain time (some number of clock cycles). To be more clear, Figure 5.5 a) shows a time compaction scheme; n input test patterns are applied to DUT, then n output patterns are taken from DUT. Those n output patterns are fed to a time comparator. Then, q output patterns are generated in the time compactor. Final result of the compaction is called as the *signature*. An LFSR is an example of time compactor. In *space compaction (SC)*, wide bit-streams from memory-under-test are compacted to a narrower bit-stream. Before the m-bit wide output pattern from DUT is fed to a time compactor, as an intermediate step, it is shortened into p-bit wide, afterwards, this shortened output patterns is fed to a time compactor, see Figure 5.5 b). A parity checker can be given as an example of space compactor.

In [59], BIST compactors were investigated in terms of time (T) vs. space (S), circuit

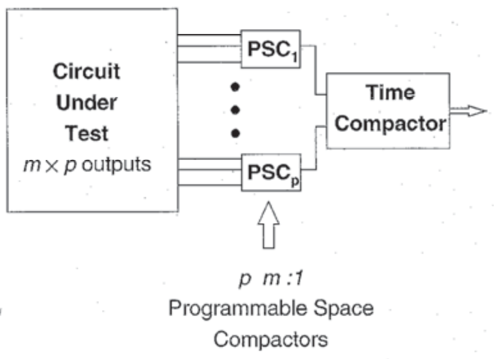


Figure 5.6: Programmable Space Compactor (PSC) based ORA of a Memory BIST [59]

function specificity (FS) vs. independence (FI), linearity (L) vs. nonlinearity (NL), combinational (C) vs. sequential (S). A taxonomy for time and space compactors used in BISTs was given. In addition, this study contributed to the literature by introducing a new compactor type: *Programmable Space Compactors (PSCs)*; a function specific, linear or nonlinear, combinational or sequential space compactor with any compaction ratio $m : p$ as shown in Figure 5.6.

Here, three well-known examples of compaction will be given [4]: transition counter, syndrome counter and signature analysis. **Transition counting** method counts the number of transitions from 0 to 1 and 1 to 0 in the output response. **Syndrome counting** method, also named as *1's counting*, counts the total number of 1s in the whole output response sequences. For multiple-output DUTs, by giving different weights to outputs, a weighted compaction is performed for both transition and syndrome counting methods [91]. **Signature analysis** is an pplication of the theory of *Cyclic Redundancy Checking (CRC)*, where the output response is compacted to a relatively shorter form which is called as the *signature*. Compaction is performed by an LFSR. LFSR should have the same-length with the output response. At the end of the test, signature is compared with a reference signature to check the correctness. Both *single-input* or *multiple-input signature analysis (MISR)* are avaliable. A MISR, also known as *multiple-input linear feedback shift register* or *parallel signal analyzer(PSA)*, is both a space and time compactor that reduces the number of outputs to be compacted [89]. After space compaction, output is further compacted by LFSR.

Although compaction is simpler than the compression, it has an aliasing probability. Research has been performed to analyze and decrease the probability of aliasing [113, 35, 87, 121, 5, 62, 31]. A final remark is that for a system, the compaction algorithm and compacted result should be simple and efficiently small in terms of area.

Last option for ORA is a microprocessor in case of an on-chip μP-based Memory BIST implementation where output results are read from the memory and compared with reference data in ALU of microprocessor.

5.3.3 Device-under-Test (DUT)

Memory BIST (MBIST) is capable of testing different types of memories. In addition, Memory BIST may be embedded into the same chip with the memory to-be tested (i.e., internal MBIST) or be a separate chip (i.e., external MBIST). Firstly, Memory BIST for different memory types will be discussed; thereafter internal versus external Memory BIST will be discussed.

5.3.3.1 Memory Types

This section discusses Memory BIST solutions for different memory types (e.g., SRAM, DRAM).

SRAM MBIST

As explained in Chapter 2, SRAMs are fast, do not require any refreshing and have a non-complex peripheral circuitry. Thus, they are an attractive candidate to be cache memories that are closely positioned to the high computation logic such as processors or hardware accelerators, and they are used for high-speed data access from several masters from their multi-ports. Therefore, a Memory BIST for SRAMs should support at-speed, multi-port, any address configuration testing. There are many MBIST solutions proposed for SRAMs [28, 82, 29, 17, 117].

DRAM MBIST

DRAMs are relatively slower than SRAMs due to their refreshment nature. They are cheaper and capable of storing huge amount of data. Hence, they act as the main data storage. When required, necessary part is captured and transfered to the local cache memories which are in type of SRAMs. Since DRAM size is high, MBIST area overhead is lower than the other memory MBISTs. However, due to their refreshment requirement, DRAM MBIST controller, specifically FSM controller, becomes more complex and slower which decrease the at-speed testing capability [95, 86]. Figure 5.7 shows an example of MBIST for DRAM. To speed-up the MBIST, a dual FSM based controller for TPG was developed. Basically, complex FSM was divided into two FSM levels which are faster than single FSM level [57].

ROM MBIST

ROMs are very important components since the initialization code of the system is stored on them. ROMs are in several types related to their programmability feature: i.e., ROMs are burned-in during the fabrication; EPROMs are electrically programmable, etc. BISTs for ROMs do not involve any write-data stepper since they are read-only memories. For ORA part of ROM MBIST, compaction is the mostly applied method (e.g., parity-based, count-based, polynomial division-based (signature analysis) compaction). Hence, a ROM is significantly crucial for a system, number of error escapes are tried to be minimized as low as possible. However, all those methods have error escapes due to the aliasing, see Section 5.3.2. In [122, 123], a new method *Exhaustive Enhanced Output Data*

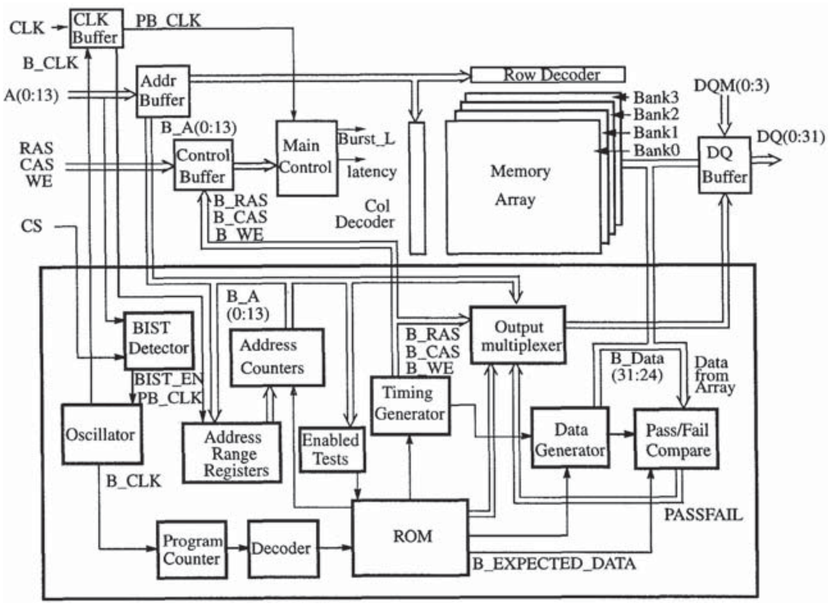


Figure 5.7: a BIST Architecture for a DRAM [86]

Modification (EEODM) was proposed to further decrease the number of error escapes. EEODM uses exhaustive address generation (i.e., binary, gray-code counters, maximum-length LFSR) for testing; it further uses signature analysis (linear) and counter-based (nonlinear) compaction in addition to ODM. As a result of decreased error escapes, fault coverage of EEODM is higher than previously proposed compaction methods of Section 5.3.2.

Register File MBIST

Register files are commonly highly embedded low-scale data storages. In [118], a case study was done for a dual port read/write register files with synchronous read capability. Since, the size of register files are in low-scale, area overhead of BIST implementation was reported between 4 to 12%, relatively higher than BISTs for RAMs or Flash.

Flash MBIST

Flash memories are becoming widespread in embedded SoCs or as commodity memories; implying that MBIST concept is inevitable for Flash memories, too. Flash memories have random-access read or write ($w0$) operations as RAMs, however they can not perform a random-access erase ($w1$), whereas only a block or whole-chip erase. Therefore, erase and program operations affect the development of March tests for Flash memories, as well as the Flash MBIST implementation.

March-like testing algorithms for Flash memories (Flash March, March FT and March 2-step) were proposed [75, 115, 12], respectively.

[115] presents a MBIST for flash memories. As mentioned before, erase and program steps dominates the test time; in this case resulting in 44.612 and 13 seconds and 3.2%

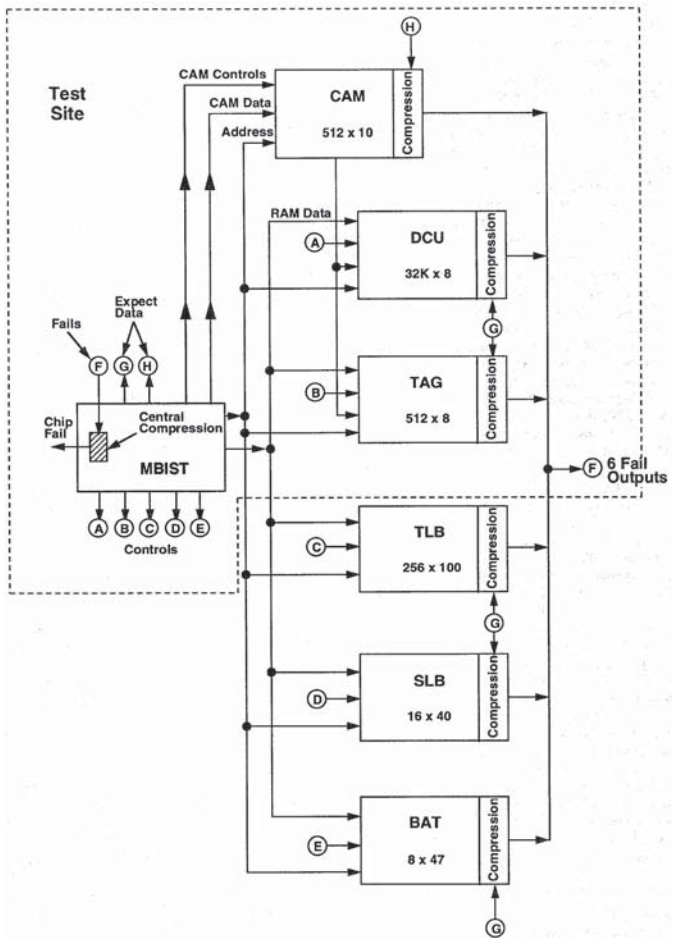


Figure 5.8: Central FSM-based BIST Engine for multiple memories [96]

and 2.28% area overheads for 512 KB embedded SoC and 128 KB commodity Flash memories, respectively. [12] presents another MBIST implementation that further decreases the area overhead, a processor-based BIST was proposed; obtaining 0.71% area overhead for a 128 KB for an embedded Flash on SoC. To sum up, the erase and program operations dominate flash memory tests and result in the range of 10 to 40 seconds test time.

5.3.3.2 Internal versus External MBIST

This section discusses Memory BIST solutions for internal and external MBIST.

Internal MBIST

In case of internal MBIST, BIST hardware may be inserted into a SoC or be a part of a distinct memory chip.

Embedded on SoC: With the decreasing process technology sizes, a SoC consists

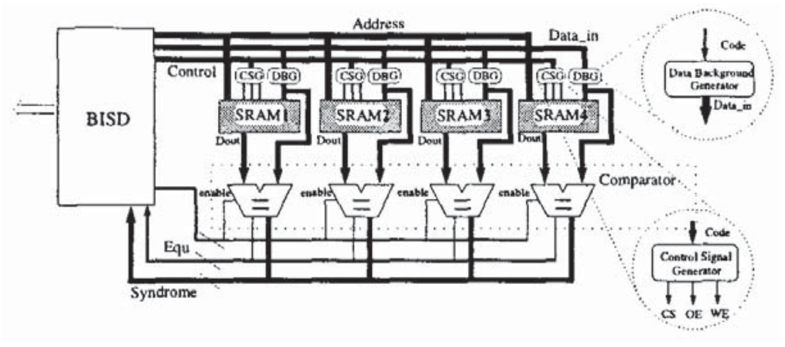


Figure 5.9: BIST scheme for heterogenous SRAM clusters [111]

of distributed memories in the order of hundreds varying in terms of their capacities, configurations, types and data widths. For example, in a large networking SoC IXE2000 [16], there are 121 memories (12 single-port, 109 dual-port) varying in size from $32x1$ to $8Kx32$ bits. This causes sequential testing where memories are combined into groups, and test procedure is performed in steps. In [96], a centralized FSM-based BIST Engine was proposed to test embedded memories in microprocessors. Figure 5.8 shows that all of the address, data and control signal generation were generated on the central BIST unit, send to distributed memories. This method [70, 96] might cause problems on signal routing to each memory separately, excessive area overhead and power consumption of long wires and crosstalk issues. A simplified solution is to move the test signal generation part of BIST close to the local memories, while a central control unit communicates with each of them by a less number of wiring or a shared test bus. In [111], a shared BIST scheme was proposed to handle heterogeneous SRAM clusters varying in memory width, latency, address space, controlling signals, number of ports, and different clock domains. Figure 5.9 shows that control and data background pattern generators were moved close to the local memories. A similar approach was adapted in [71], a BIST architecture for distributed memories varying in sizes was proposed with the difference of distributed wrappers per each memory. Figure 5.10 a) shows a central BIST controller implemented as a processor which reads the instructions from a ROM; wrappers surrounding each SRAM consisting address, control, background generators and a comparator; communication signaling between central BIST processor and distributed wrappers; and a scan chain connecting all wrappers for the diagnosis report. Instruction $CONF$ is used to select which memories will be tested. Each time, a test primitive (i.e., $w0, r1, ...$) is read from central BIST processor memory and sent to all wrappers, and when all of the selected memories finish to perform the test primitive, a new test primitive is sent. After performing all test primitives, results are read by processor. Figure 5.10 b) shows that wrappers might be implemented for a single as well as a group of RAM cluster. Although, each wrapper has its own address (AG), data background pattern (BPG) and control generators, only a 1.06% total area overhead was reported, see Figure 5.11. Moreover, linear relations between BPG, AG areas vs. memory word width and addressing space of memories were graphed.

An important issue related to the testing of distributed memories on an embedded

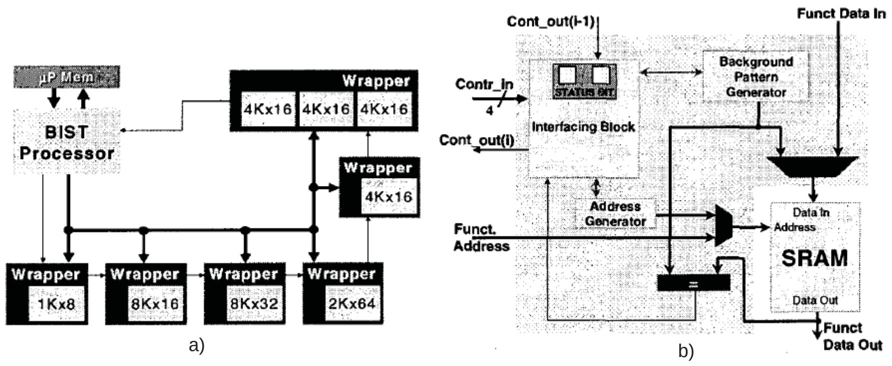


Figure 5.10: Architecture of a) a distributed MBIST and b) a single wrapper [71]

RAM	*RAM Area*	*Wrapper Area*	*Overhead*
1Kx8	25,831	1,788	6.92%
4Kx16	139,740	2,291	1.64%
8Kx16	265,355	2,347	0.88%
2Kx64	314,262	3,653	1.16%
3*[4Kx16]	419,220	3,254	0.78%
8Kx32	492,537	2,908	0.59%

a)

Total RAM area	1,656,945
Total Wrapper area	16,241
BISTprocessor area	1,392
Total	1,674,578
Total area overhead	**1.06%**

b)

Figure 5.11: Area report for a distributed BIST architecture [71]

SoC, is the excessive power release. During testing mode, power release becomes higher than the normal system activity, moreover since several memories are being tested simultaneously, this might result in a system damage. In [119], a central BIST scheduler that takes into account this issue, was proposed. Also, in [21], rearrangement of memory test algorithms was suggested based on single bit change counting (SBC) method, resulting in lower heat dissipations varying from a factor of 2 to 16. For the distributing memory testing, test scheduling and grouping of memories should be decided well to decrease area overhead by resource sharing, avoiding hazardous power rates and shortening the test length.

Commodity: Commodity memories are the distinct external memories having the capability of high-scale memory capacity. A BIST can be involved in their memory chip package to test run-time on-field errors. Since they are large scale data storages, area overhead of MBIST is small. In addition, they are single memories, MBIST implementation is relatively simpler than embedded SoC or external BISTs.

5.3.3.3 External MBIST

Obviously, all ICs including memory chips are tested during the production process before sent to customers; they are called known-good dies (KGDs). However, errors can still appear during run-time operations inside of the memory or on the board level as a result of wear-out process. In Table 5.3, the second column shows the numbers of I/O pins for 8 different systems, and the third column shows the numbers of pins per system

Table 5.3: Interconnection percentages between external memory and system [63]

	Total Functional I/Os	Interconnections with Ext. Mem	%	Types of Ext. Mem
Chip 1	539	68	13%	SRAM
Chip 2	1382	1290	93%	D-DDR,QDR,FCRAM
Chip 3	1398	1047	75%	D-DDR,QDR,FCRAM
Chip 4	1448	1288	89%	DDR,QDR,FCRAM
Chip 5	1470	1154	79%	DDR,QDR,FCRAM
Chip 6	752	75	10%	DDR
Chip 7	611	102	17%	QDR,RLDRAM
Chip 8	804	301	37%	RLDRAM

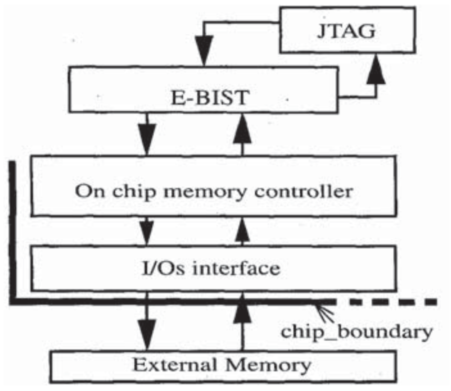


Figure 5.12: High level External BIST scheme [20]

and memory pairs. As seen in fourth column, a high percentage of system's I/O pins are dominated by the memory connections. Moreover, IEEE STD 1149.1 Boundary-scan standard can not test the interconnections at a high speed. Thus, run-time testing of both memory and interconnects after assembly step gains more importance. By locating a MBIST embedded inside of the SoC to test an external memory, and optionally the interconnections, this problem can be solved. This type of MBIST is called as an *External memory BIST* (EBIST). An EBIST is capable of at-speed testing of the content of memory and the interconnects for DC and AC faults [20]. However, memories highly vary in terms of their speed, data rates, burst rates, number of clock cycles, size configurations, etc. Hence, it is not easy to support all of those complex features by the same EBIST. [20] suggests a novel solution: as shown in Figure 5.12, EBIST uses the existing memory

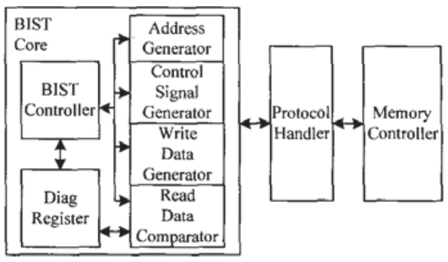


Figure 5.13: Protocol Handler for external MBIST [63]

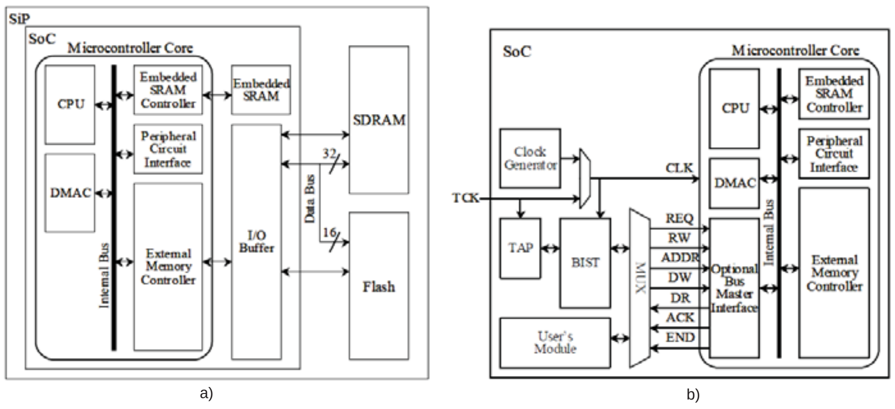


Figure 5.14: SoC architecture a) before and b) after external BIST insertion [114]

controller on the system for the memory access. Thus it saves the area overhead of memory access protocol implementation. Memory controller handles timing and data-type conversion between system and the external memory. EBIST is configured via JTAG TAP controller through the standard and low number of pins for configuration.

Due to the different memory types, memory controllers vary, too. Memory controllers can be classified into three groups as: memory controller with a fixed latency, non-fixed latency (handshaking), both fixed-latency and handshaking protocols [63]. Figure 5.13 [63] shows a solution with an intermediate configurable protocol handler between EBIST and the memory controller. Protocol handler is configured regarding to the memory controller latency, thus all different type of memories are supported.

Figure 5.14 a) [114] shows a typical SoC architecture with CPU, system bus, etc. for SDRAM and superAND Flash memories on a System-in-Package (SiP). Figure 5.14 b) [114] shows the SoC after EBIST insertion to the SoC. The difference of this example with the previous studies is that EBIST does not directly connected to the existing memory controller, instead it is connected to an optional bus master interface. Through the bus master interface, BIST Engine has an access to the SoC internal bus and some specific address registers on the CPU that are controlling clock generator and memory controller. In configuration mode, TCK; in test mode, high frequency signal from clock generator is chosen for at-speed test. The area overhead of this design was reported as 1.7% of the SoC logic.

5.4 Test Architecture

Accessing to the memory cell arrays can be done in any combination of single to multiple arrays and single to multiple bits. A taxonomy was developed to classify the MBIST architectures [33, 100]; here they are explained briefly.

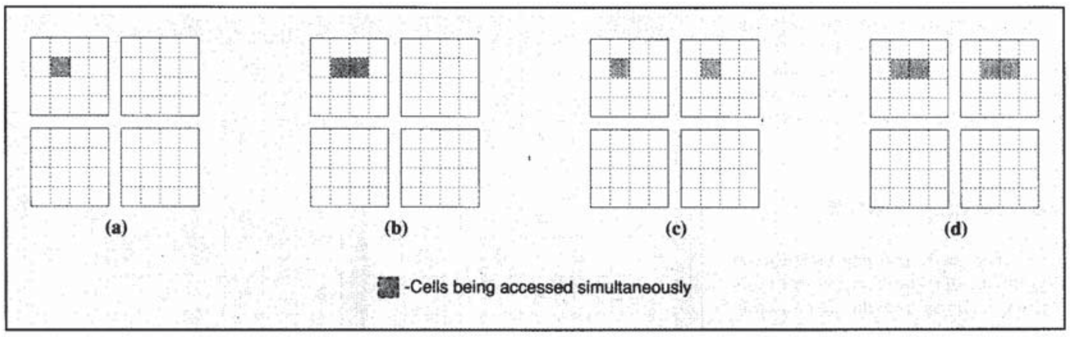


Figure 5.15: Memory cell array accessing schemes for a) single-array single-bit, b) single-array multiple-bit, c) multiple-array single-bit and d) multiple-array multiple-bit [33]

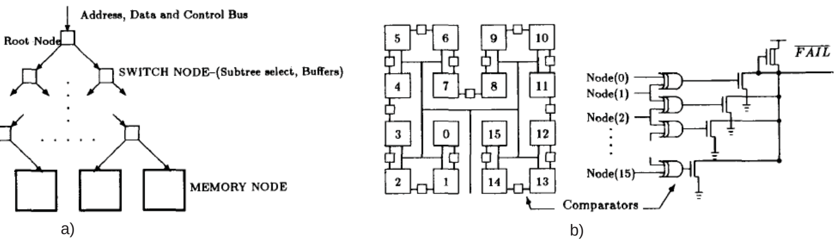


Figure 5.16: a) TRAM and b) mutual comparison of multiple bits [60]

Single Array Single Bit (SASB)

Figure 5.15 a) shows that a single bit of a single memory cell array is tested at a time. Thus, SASB has the longest test time and high fault coverage. SASB is also called as normal-mode accessing.

Single Array Multiple Bit (SAMB)

Figure 5.15 b) shows that the multiple bits of a single memory cell array are tested at a time. Generally, same bits on the same row are accessed to be faster instead of accessing to bits on the same column. Test time is shortened as the number of multiple bits accessed simultaneously. In case of a word access (whole line), speed-up is equal to $\sqrt{n}$, where n is the the number of bits in memory cell array under the assumption of square layout. However, it requires some additional hardware on address decoder for multiple bit access. SAMB is also called as line-mode (word) accessing. Fault coverage is not as high as SASB due to the restriction of coupling fault detection from same row.

Multiple Array Single Bit (MASB)

Figure 5.15 c) shows that a single bit of multiple memory cell arrays are accessed at a time. Test time is shortened as the number of parallel multiple memory arrays. Figure 5.16 a) [60] shows a tree RAM (TRAM) architecture with parallel memories per level. In addition, instead of a direct comparator, a mutual comparator can be applied where

Table 5.4: Advantages and drawbacks of the MBIST implementation types

	Test Time	Area OH	Routing OH	Flexibility
Hardwired	short	low	high	zero
Microcode/(Processor)	average	high	low	low
On-chip Processor	long	zero	zero	high

the single bits from multiple arrays are compared within each other as shown in Figure 5.16 b) [60]. MASB has a high fault coverage.

Multiple Array Multiple Bit (MAMB)

Figure 5.15 d) shows that the multiple bits of multiple memory cell arrays are accessed at a time. Test time is shortened as the number of parallel multiple memory arrays times multiple bits accessed. Fault coverage is limited due to the same reason as SAMB.

5.5 Implementation of MBIST

Design choices for a MBIST implementation will be discussed in terms of MBIST flexibility, insertion, interface to memory, diagnosis and repair capability.

5.5.1 Flexibility

The core of a BIST engine can be realized in two ways:

1. Fixed method: where the test generation capability of MBIST is highly limited due to the finite state machine implementation,
2. Programmable method: where the test generation capability of MBIST has a higher flexibility due to the microcode (processor) or on-chip processor implementations.

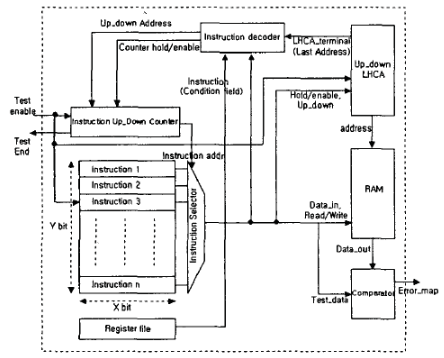
Table 5.4 shows a brief comparison of the MBIST implementation types. They will be explained in detail.

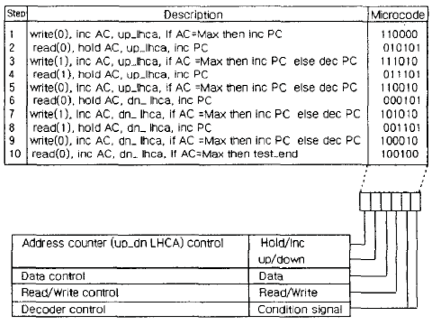
Finite state machine MBIST

Required address, data and control signals are generated by a finite state machine (FSM). March elements are implemented as states. From one March test to several number of March tests might be implemented. However, FSM based MBIST is not flexible enough. Once, it is implemented in hardware, reconfiguration can be done between several test algorithms which are already built-in.

Microcode based MBIST

To solve the flexibility problem, test algorithms are compiled into instructions and then saved in programmable memories as shown in Figure 5.17 a) [116]. Figure 5.17 b) [116] shows a March test realization for the proposed MBIST. Each bit position in the microcode corresponds to a function in the MBIST. When a new algorithm is developed, it can be loaded into the programmable memory without any change in hardware. However,





Step	Description	Microcode
1	write(0), inc AC, up_lhca, If AC=Max then inc PC	110000
2	read(0), hold AC, up_lhca, inc PC	010101
3	write(1), inc AC, up_lhca, if AC =Max then inc PC else dec PC	111010
4	read(1), hold AC, up_lhca, inc PC	011101
5	write(0), Inc AC, up_lhca, if AC =Max then inc PC else dec PC	110010
6	read(0), hold AC, dn_ lhca, inc PC	000101
7	write(1), inc AC, dn_ lhca, if AC =Max then inc PC else dec PC	101010
8	read(1), hold AC, dn_ lhca, inc PC	001101
9	write(0), inc AC, dn_ lhca, if AC =Max then inc PC else dec PC	100010
10	read(0), inc AC, dn_ lhca, If AC=Max then test_end	100100

Address counter (up_dn LHCA) control	Hold/inc up/down
Data control	Data
Read/Write control	Read/Write
Decoder control	Condition signal

a) b)

Figure 5.17: a) Microcode-based MBIST architecture and b) a March test realization [116]

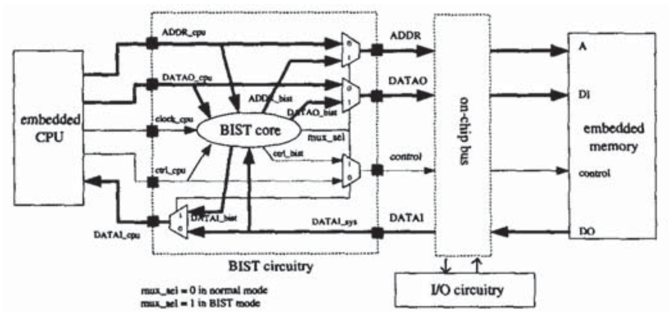


Figure 5.18: On-chip 6502 processor-based BIST architecture [97]

microcode based MBIST brings an area overhead of the instruction storage, instruction decoding logic and the program counter (PC). Thus, occupied area becomes larger compared to the FSM based MBIST. An important remark is that sometimes microcode based MBIST is called as a processor based MBIST, since this method also has the instruction memory, decoder and program counter, etc. [27].

On-chip processor based BIST

This is the case where an existing on-chip processor is used for the memory test. This method has the highest flexibility. Test algorithms are written in assembly-language program, and then through the system bus, or maybe NoC in future, read-/write sequences are applied to the memory. Since the commands are sent via the system bus, on-chip processor based MBIST implementation has a high test time. For the implementation in Figure 5.18, 9.6 seconds test time was reported in [97], which consists of a 6502 MOS Technology processor implementing March C- algorithm, $\{\Updownarrow(w0); \Uparrow(r0w1); \Uparrow(r1w0); \Downarrow(r0w1); \Downarrow(r1w0); \Updownarrow(r0)\}$. On the other hand, microcode based MBIST proposed in [56] reports only 0.4 seconds for the same test algorithm. Figure 5.19 [93] shows an on-chip ARM processor based BISR architecture. Test signal generators and output response analyzer are distributed to the memory wrappers. By

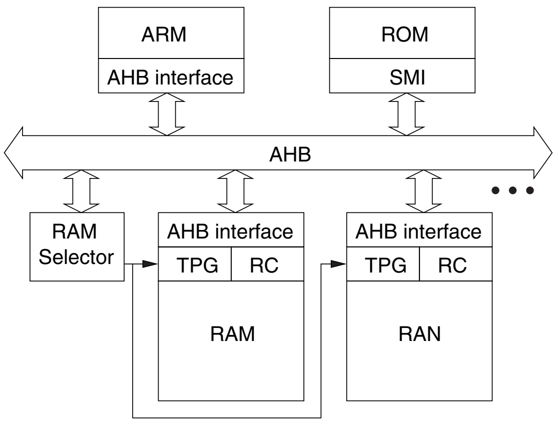


Figure 5.19: On-chip ARM processor-based BIST architecture [93]

programming the central ARM processor, test commands are sent through the system bus to the memory wrapper.

5.5.2 Insertion

MBIST can be used to test only a single memory (i.e., private MBIST) as well as a group of memories (i.e., shared MBIST) in a system.

Shared MBIST

Shared BIST is preferred in case of testing several memories to decrease the area overhead; mostly for the distributed memories on embedded SoCs or for the different type external memories, see Section 5.3.3.2 and 5.3.3.3. Only control unit or processor; or all (control, TPG and ORA) sub-blocks in MBIST can be shared by different memories. To further decrease the area overhead, TPG and ORA might be shared, however that complicates the wire routing. In shared MBIST implementations, interface connection between memories and MBIST can be parallel as in Figure 5.9 and 5.8 or can be serial. Both testing methods (i.e., sequential or parallel) and testing interface (i.e., serial or parallel) are orthogonal to each other; meaning that any combination can be chosen, (e.g., parallel testing with serial interface, sequential testing with parallel interface). Interface types between MBIST and memory will be introduced in next section. Here sequential and parallel testings will be explained. To simplify the test process, memories can be grouped and tested in two ways in case of shared BIST implementation as [78]:

1. **Sequential testing:** When the number of memory groups is more than 1 (i.e., memories are separated in several groups), all memories in group 1 are tested, then all memories in group 2, and so on. Number of memories in a group can vary from 1 to any number. Sequential testing saves the area overhead by sharing the hardware resources. For example, a same comparator can be used for whole memories to the extent of test time as shown in Figure 5.20 a) [78]. Number of comparators is determined by the maximum number of memories in a group. A

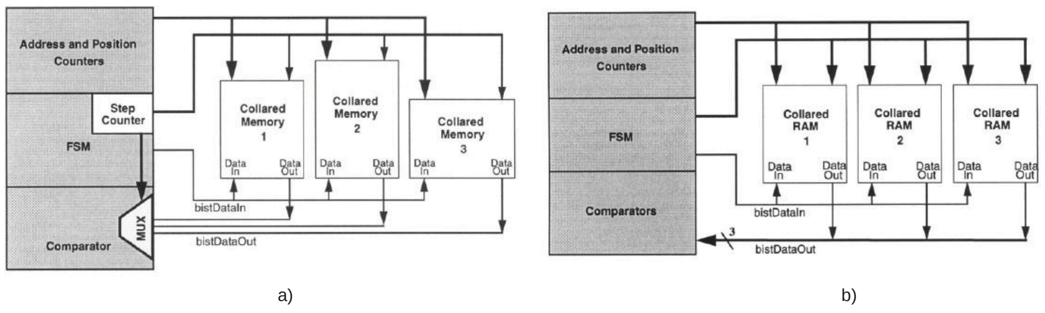


Figure 5.20: Shared MBIST schemes for a) sequential and b) parallel testing [78]

remark is that all memories in a group use same algorithm with same operations and number of bits per word should be same.

2. **Parallel testing:** When the number of memory groups is 1 (i.e., whole memories are collected in a single group), ORA can not be shared; resulting in a high area overhead. Figure 5.20 b) [78] shows that each memory requires its own comparator. In addition, parallel testing can be applied to speed up the test process of a single memory. Once the memory cell array is partitioned into subgroups, multiple arrays can be tested simultaneously [78]. For shared MBIST parallel testing, memories should be highly similar or extra hardware should be designed to take care of the addressing of memories in different sizes.

Private MBIST

Private MBIST is preferred to test commodity memories where memory and BIST hardware are combined in the same package. Since only one memory has to be handled, MBIST architecture is optimized. However, the area overhead is still higher than the shared MBIST.

5.5.3 Interface to the memory

MBIST can be connected to the memory serially as well as in parallel. Each will be discussed in detail including advantages and drawbacks.

Serial Interface

In serial interfacing, test data is sent to the memory in a serial way. Serial interfacing simplifies the interconnection routing, since the amount of wiring between MBIST and the memory is decreased. In case of high data widths, area saving is high. For serial interfacing, a serial to parallel converter buffer (i.e, a scan chain) is located at the I/O ports of the memory.

The serialization process affects the test algorithm development and test time, but not the fault coverage. Serialized March algorithms are applied such as SMarch, SGalpat, SWalk [76]. Figure 5.21 [76, 77] shows the proposed serial interfacing architecture. Each

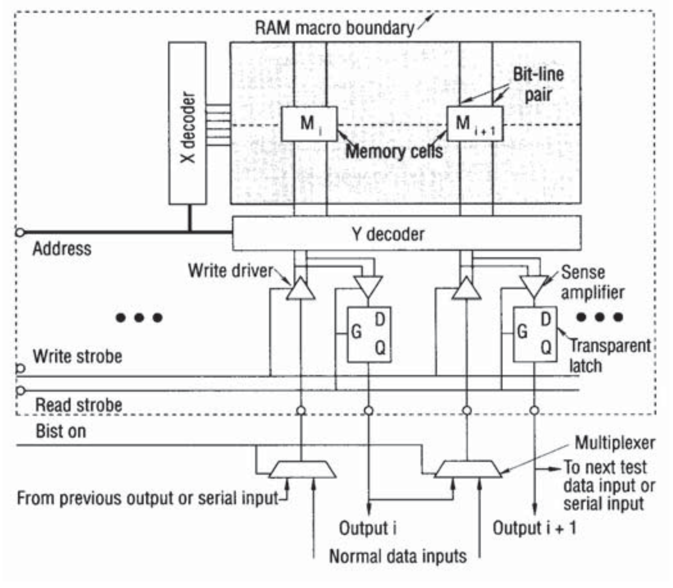


Figure 5.21: Serial interfacing with unidirectional I/O ports [76]

data input is connected to the previous input's data output through a multiplexer. However, pushing test data in and pushing output response out take c clock cycles, where c is the memory data width. To further speed up this bottleneck, [110] proposes a novel solution: test data is delivered from the MBIST to the memory serially, then converted to parallel through the *Serial to Parallel Converters (SPCs)*. Once the test is performed, the output data is analyzed in parallel by *Local Response Analyzer (LRA)* (e.g., comparator). Later on, a single-bit result of LRA per memory is sent to a global MISR. Finally, the signature is sent to MBIST serially to be evaluated.

An important issue is that serial interfacing might cause to the *serial fault masking* problem and only avoided by a *bidirectional serial interface* [10].

Serial interfacing achieves a low area usage compared to parallel interfacing; especially in case of high data widths. On the other hand, it sacrifices from the longer testing time, and the modification of well-known algorithms for serial interfacing and more complex control unit.

Parallel Interface

Parallel Interfacing uses the original I/O ports of memory. Data patterns are directly applied without any serialization as shown in Figure 5.22. Thus, the test time is not as high as serial interfacing. In addition, the nonexistence of serialization process decreases the complexity of MBIST. Still, depending on the number of data lines between MBIST and the memory, area overhead might be costly.

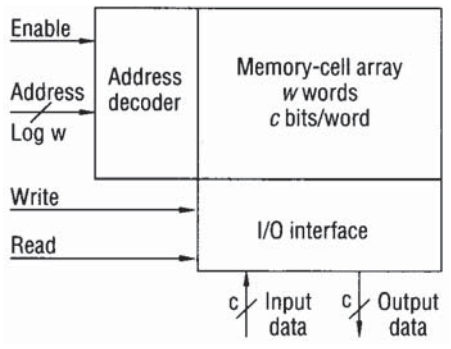


Figure 5.22: Parallel interfacing I/O ports [76]

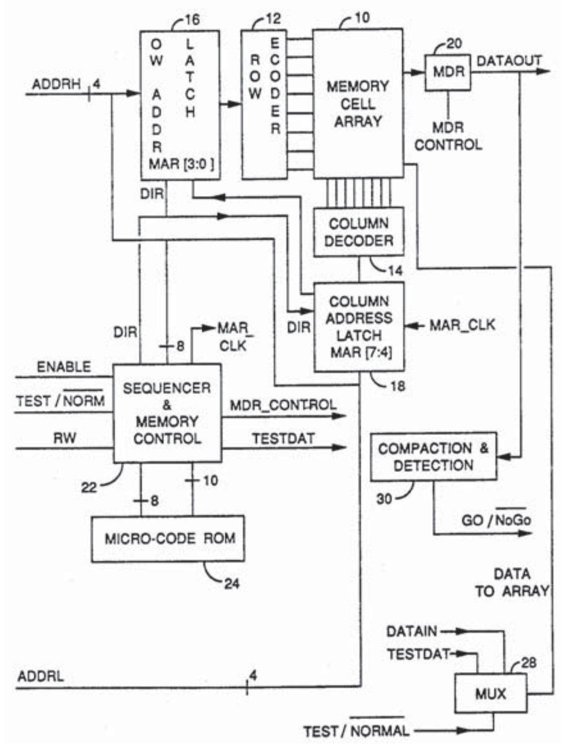


Figure 5.23: MBIST with only detection capability [84]

5.5.4 Capability

This section discusses the test only, diagnosis and repair capabilities of built-in self-testing.

Detection Only

When a MBIST engine is only capable of determining whether a memory is fault-free or not, then it is called as a BIST with detection only. Figure 5.23 shows a MBIST that generates only the $\overline{FAIL}$ /$FAIL$ or $\overline{NoGO}$ /GO signals. BIST is applied with

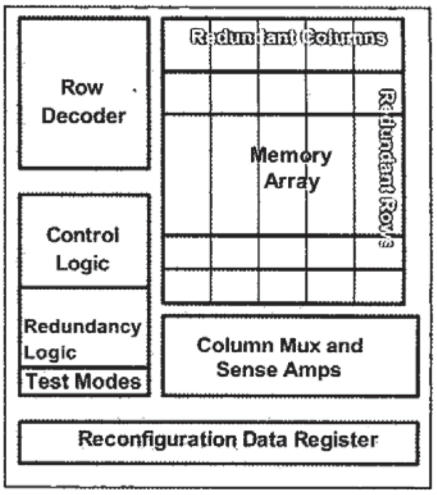


Figure 5.24: Memory with redundant rows and columns [120]

the memories without redundant spare memory cell allocations. For example, in case of small buffers, addition of a redundant memory blocks and repair block hardware are costly in terms of area. BIST neither reports the location nor repairs the faulty cell. On the other hand, shrinking process sizes lower the memory yield. Therefore, memory diagnosis and repair becomes a must.

Diagnosis and Repair

When a BIST engine is capable of diagnosing the fault and handling the repair mechanism, it is a called as *Built-In Self-Diagnosis (BISD)* or *Built-In Self-Repair (BISR)*. Generally, BISD module detects and stores the address of the faulty cells. Then, the redundancy allocation (RA) is handled by BISR module. Figure 5.24 shows the extra elements for BISR such as: redundant memory elements (e.g., rows, columns), repair logic, address storage of the faulty cell. The basic idea behind the self-repair is mapping the faulty cells to the redundant memory rows/columns. To be efficient in BISR mapping, a redundant row/column should cover the faulty cells as much as possible. Naturally, BISD or BISR has a higher area overhead than BIST. In case of high number of memories, addition of a BISR capability to each memory might be high costly.

Generally in BISR, while one register is storing the faulty address, a second one stores the mapped row/column address of redundant memory, then in case of an access to the faulty region, addresses are translated. However, [15] proposes that this method is very expensive in terms of register area and signal routing overhead. Proposed solution is using the content addressable memories (CAMs) with on-line address remapping. It has two mode of operations: *SR_only/BISTAR*. In BISTAR mode, both self-testing and repairing are active, where in SR_only mode, only self-repairing is performed and new faults are not handled.

Another good example is the Self-test and Repair (STAR) processor that is capable of BIST, BISD and BISR functions [120]. STAR processor is combined with the memory

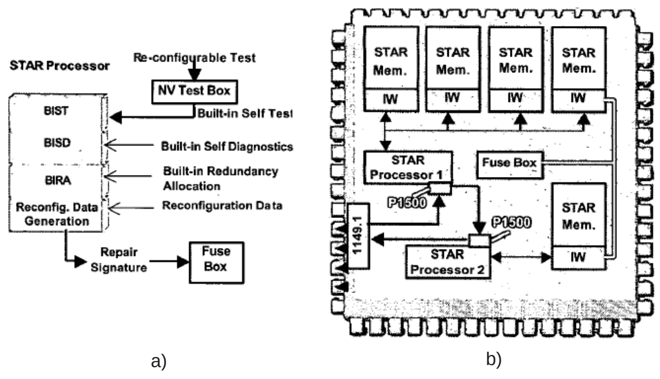


Figure 5.25: (a) STAR processor and (b) a SoC solution [120]

and called as the *smart memory*. Figure 5.25 shows the architecture of a smart memory: distributed memories are grouped depending on their similarities; each group is under the control of a STAR processor. Electronically programmable fuses (eFUSEs) are used to reconfigure the faulty cell address to a fault-free redundant memory element. by programming the .

To note that, mapping of the faulty memory cells to the spare rows/columns is an NP-hard problem [64], hence the solution requires highly efficient heuristics. Some research on spare allocation for BISR is done in [68, 67].

5.6 Highly appreciated MBIST features

In general, a state-of-the-art MBIST can be summarized as *"low cost high quality memory testing"*. This section presents the highly appreciated features [29] of a MBIST.

For high quality memory testing, MBIST should be flexible, programmable, orthogonal, capable of nested looping and at-speed testing. **Flexibility** refers to the support of different memory test families (e.g., linear: March, Scan and nonlinear: Hammer, GalPat), since they target different fault types. As a result **nested loops** should be supported. In addition, a MBIST should be **programmable**; enabling user-defined test algorithms and in-the-field programming. Test generator blocks should be **orthogonal**; they should generate any combination of memory addressing, data background pattern and memory operation without limiting each other. In addition, MBIST should be capable of **at-speed testing** to detect the faults that only occur at the operational frequency of the memory.

Furthermore, MBIST should have a **low area overhead** for low cost and easy routing. It should have a **generic and modular** architecture for any memory type, size configuration and future extensions. Then, the time-to-market is shortened and the development of new extensions is simplified. Moreover, MBIST should be able to test **distributed memories** to manage today's and tomorrow's SoC systems with hundreds of memories. Lastly, without **diagnosis and repair** capabilities, it is impossible to catch high memory yield and high profit [124].

Generic and Orthogonal March Element based Memory BIST 6

This chapter introduces a new memory BIST based on a generic and orthogonal March element concept. This chapter is organized as follows. Section 6.1 gives a list of widely used memory test algorithms in industry. This list will be inspected to derive the memory BIST requirements in Section 6.2. Section 6.3 provides the concept of the Generic March Element (GME) and shows how this concept can be used to generate any targeted algorithm; an optimal list of generic March elements will be derived based on the list of memory test algorithms of Section 6.1. Section 6.4 provides the high level architecture of GME based memory BIST, including the BIST register and the command sets. Section 6.5 illustrates how the command set can be used to generate the test algorithms.

6.1 Targeted memory test algorithms

As mentioned in Chapter 4, memory test algorithms can be classified into two groups [7]. The first group consists of the historically traditional tests that were developed intuitively, such as Scan, GalPat, Walking 1/0, etc. The second group consists of tests that are systematically developed based on the fault model or fault primitive concept, such as March SS, March SL, March RAW, etc. Industrial results [49, 7] show that tests with high fault coverage (FC) in theory also have high FC in practice. Therefore, in this study a BIST Engine which is able to support the memory test algorithms with high FC and unique fault detection was aimed.

Table 6.1 lists a set of the most popular and widely used memory test algorithms in industry. For March notation and the explanation of each algorithm, see Chapter 4. Table 6.1 clearly shows that algorithm 1 to 15 require similar addressing (e.g. $\Uparrow$ or $\Downarrow$). On the other hand, algorithm 17 to 18 require special addressing and/or operations. For example GalRow performs operation along rows, GalCol along columns; algorithm 16 applies with the Address Complement (AC) $\Uparrow^{AC}$ addressing; MOVI requires special 2^i counting method; HamRh applies with the Hamming operations, etc.

6.2 Memory BIST requirements

In order to be able to perform the algorithms listed in Table 6.1, the memory BIST has to satisfy many requirements and support different features. These are either related to the operations of the algorithm itself, to the address generation or the data generation. In this section, we will restrict ourself to the address and the data generations. In Section 6.3 we will provide more about operations.

Table 6.1: Set of candidate algorithms

#	Name	B(GME#)	Description
1	Scan [1]	0(0,1)	$\{\Downarrow(w0);\Uparrow(r0);\Uparrow(w1);\Downarrow(r1)\}$
2	MATS+ [1]	0(0,2)	$\{\Updownarrow(w0);\Uparrow(r0,w1);\Downarrow(r1,w0)\}$
3	March C- [100]	0(0,1,2)	$\{\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);\Downarrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0)\}$
4	March C+ [117]	0(0,1,3)	$\{\Updownarrow(w0);\Uparrow(r0,w1,r1);\Uparrow(r1,w0,r0);\Downarrow(r0,w1,r1);\Downarrow(r1,w0,r0);\Downarrow(r0)\}$
5	PMOVI [23]	0(0,3)	$\{\Downarrow(w0);\Uparrow(r0,w1,r1);\Uparrow(r1,w0,r0);\Downarrow(r0,w1,r1);\Downarrow(r1,w0,r0)\}$
6	March B [94]	0(0,4,5,6)	$\{\Updownarrow(w0);\Uparrow(r0,w1,r1,w0,r0,w1);\Uparrow(r1,w0,w1);\Downarrow(r1,w0,w1,w0);$ $\Downarrow(r0,w1,w0)\}$
7	Alg. B [73]	0(0,6,7)	$\{\Updownarrow(w0);\Uparrow(r0,w1,w0,w1);\Uparrow(r1,w0,r0,w1);\Downarrow(r1,w0,w1,w0);$ $\Downarrow(r0,w1,r1,w0)\}$
8	March G [99]	0(0,3,4, 5,6)	$\{\Updownarrow(w0);\Uparrow(r0,w1,r1,w0,r0,w1);\Uparrow(r1,w0,w1);\Downarrow(r1,w0,w1,w0);$ $\Downarrow(r0,w1,w0);Del100;\Updownarrow(r0,w1,r1);Del100;\Updownarrow(r1,w0,r0)\}$
9	March U [102]	0(0,2,7)	$\{\Updownarrow(w0);\Uparrow(r0,w1,r1,w0);\Uparrow(r0,w1);\Downarrow(r1,w0,r0,w1);\Downarrow(r1,w0)\}$
10	March LR [103]	0(0,1,2,7)	$\{\Updownarrow(w0);\Downarrow(r0,w1);\Uparrow(r1,w0,r0,w1);\Uparrow(r1,w0);\Uparrow(r0,w1,r1,w0);\Updownarrow(r0)\}$
11	March LA [104]	0(0,1,8)	$\{\Updownarrow(w0);\Uparrow(r0,w1,w0,w1,r1);\Uparrow(r1,w0,w1,w0,r0);\Downarrow(r0,w1,w0,w1,r1);$ $\Downarrow(r1,w0,w1,w0,r0);\Downarrow(r0)\}$
12	March SS [47]	0(0,1,9)	$\{\Updownarrow(w0);\Uparrow(r0,r0,w0,r0,w1);\Uparrow(r1,r1,w1,r1,w0);\Downarrow(r0,r0,w0,r0,w1);$ $\Downarrow(r1,r1,w1,r1,w0);\Updownarrow(r0)\}$
13	March RAW [42, 51]	0(0,1,10)	$\{\Updownarrow(w0);\Updownarrow(r0,w0,r0,r0,w1,r1);\Updownarrow(r1,w1,r1,r1,w0,r0);$ $\Updownarrow(r0,w0,r0,r0,w1,r1);\Updownarrow(r1,w1,r1,r1,w0,r0);\Updownarrow(r0)\}$
14	March SR [48]	0(0,7,13)	$\{\Downarrow(w0);\Uparrow(r0,w1,r1,w0);\Uparrow(r0,r0);\Uparrow(w1);\Downarrow(r1,w0,r0,w1);\Uparrow(r0,r0)\}$
15	BLIF [107]	0(0,11)	$\{\Updownarrow(w0);\Uparrow(w1,r1,w0);\Uparrow(w1);\Uparrow(w0,r0,w1)\}$
16	RaW-AC [44]	0(0,2)	$\{\Updownarrow(w0);\Uparrow^{AC}(r0,w1);\Uparrow^{AC}(r1,w0);\Downarrow^{AC}(r0,w1);\Downarrow^{AC}(r1,w0)\}$
17	GalPat [18]	1(0,2)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Uparrow_{-v}(r0,r1_v),w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Uparrow_{-v}(r1,r0_v),w1_v)\}$
18	GalRow [18]	1(0,3)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Uparrow_{R-v}(r0,r1_v),w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Uparrow_{R-v}(r1,r0_v),w1_v)\}$
19	GalCol [100]	1(0,3)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Uparrow_{C-v}(r0,r1_v),w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Uparrow_{C-v}(r1,r0_v),w1_v)\}$
20	Gal9R [106]	0(0,12)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Box(r0,r1_v),w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Box(r1,r0_v),w1_v)\}$
21	Butterfly [100]	1(0,4)	$\{\Updownarrow(w0);\Uparrow(w1_v,\Uparrow_{BF}(r0,r1_v),w0_v);\Updownarrow(w1);\Uparrow(w0_v,\Uparrow_{BF}(r1,r0_v),w1_v)\}$
22	Walk 1/0 [100]	1(0,5)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Uparrow_{-v}(r0),r1_v,w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Uparrow_{-v}(r1),r0_v,w1_v)\}$
23	WalkRow [100]	1(0,6)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Uparrow_{R-v}(r0),r1_v,w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Uparrow_{R-v}(r1),r0_v,w1_v)\}$
24	WalkCol [100]	1(0,6)	$\{\Updownarrow(w0);\Uparrow_v(w1_v,\Uparrow_{C-v}(r0),r1_v,w0_v);\Updownarrow(w1);\Uparrow_v(w0_v,\Uparrow_{C-v}(r1),r0_v,w1_v)\}$
25	HamWh [106, 98]	1(0,7)	$\{\Updownarrow(w0);\Uparrow(r0,w1^h,r1);\Uparrow(r1,w0^h,r0);\Uparrow(r0,w1^h,r1);\Uparrow(r1,w0^h,r0)\}$
26	HamRh [106, 98]	1(0,8)	$\{\Updownarrow(w0);\Uparrow(r0,w1,r1^h,r1);\Uparrow(r1,w0,r0^h,r0);\Uparrow(r0,w1,r1^h,r1);$ $\Uparrow(r1,w0,r0^h,r0)\}$
27	HamWDhrc [106, 98]	1(0,9)	$\{\Updownarrow(w0);\nearrow(w1^h_v,\Uparrow_{R-v}(r0),r1_v,\Uparrow_{C-v}(r0),r1_v,w0_v);$ $\Uparrow(w1);\nearrow(w0^h_v,\Uparrow_{R-v}(r1),r0_v,\Uparrow_{C-v}(r1),r0_v,w1_v)\}$
28	HamWDhc [106, 98]	1(0,10)	$\{\Uparrow(w0);\nearrow(w1^h_v,\Uparrow_{C-v}(r0),w0_v);\Uparrow(w1);\nearrow(w0^h_v,\Uparrow_{C-v}(r1),w1_v);$
29	MOVI [23]	0(0,3)	$\{i_0^{N-1}[\Downarrow^i(w0);\Uparrow^i(r0,w1,r1);\Uparrow^i(r1,w0,r0);\Downarrow^i(r0,w1,r1);\Downarrow^i(r1,w0,r0)]\}$
30	DELAY [1]	0(0,2,14)	$\{\Updownarrow(w0);Del50;\Uparrow(r0,w1);Del50;\Downarrow(r1,w0)\}$

$\Uparrow^{AC}$: Address Complement addressing	$\Box$: 8 neighbour cells	N: number of memory address bits
$\Uparrow^i$: 2^i addressing		k: Butterfly maximum distance
$\Uparrow_v$: all cells except the v-cell		$Del50$ and $Del100$: 50 and 100 ms delay elements
$\Uparrow_{R-v}$: all cells in the same row as the v-cell, except the v-cell		$\nearrow$: all cells on the main diagonal
$\Uparrow_{C-v}$: all cells in the same column as the v-cell, except the v-cell		h: number of Hammer operations
$\Uparrow_{BF}$: cells with a distance of 2^k to the North, East, South and West of the v-cell		

Address Generation

The memory BIST has to support different address orders (AO), different address directions (AD), different counting methods (CM); they are explained next.

1. *Address Order (AO)*: determines that the way one proceeds to the next address, which can be an increasing AO (e.g., increasing from address 0 to $n-1$; $n =$ the # of addresses), denoted by $\Uparrow$ symbol, or a decreasing AO, denoted by $\Downarrow$ symbol, and which is the exact inverse of the $\Uparrow$ AO. For example, Scan test (see Table 6.1) requires both $\Uparrow$ and $\Downarrow$ orders. When the AO is irrelevant, the symbol $\Updownarrow$ will be used. Moreover, an index can be added to the AO symbol such that the addressing range

can be specified explicitly. For example '$\Uparrow_{i=0}^{n-1}$' denotes: increase the addresses from cell 0 to cell $n-1$. In addition, the address to which the operation is applied can be specified explicitly by subscripting the operation. For example, $w0_i$ means write 0 into address i.

2. *The Address Direction (AD)*: specifies the direction of the address sequence and consists of three types: *Fast-row*, *Fast-column* and *Fast-diagonal*, which increments or decrements the row address (column address, diagonal address) most frequently. To specify the Fast-row, Fast-column and Fast-diagonal AD, the subscripts r, c and d are used with the AO respectively; e.g., $_r \Uparrow$ indicates $\Uparrow$ AO with the Fast-row AD. For example, GalRow requires $_c \Uparrow$; GalCol requires $_r \Uparrow$; HamWDhrc requires $_d \Uparrow$, see Table 6.1.

3. *The Counting Method (CM)*: determines the address sequence. Many CMs exist; e.g., there are 3!=6 ways of address counting for a 3-address memory: 012, 021, 102, 120, 201 and 210. It has been shown that the CM is important for detecting *Address Decoder Delay Faults* (ADDFs) [85, 65, 44, 26]. The most common CM is the *Linear* CM, denoted by the superscript 'L' of the AO (e.g., $^L \Uparrow$), where L specifies the address sequence 0,1,2,3, etc. Because it is the default CM, the superscript 'L' is often deleted.

   Another CM is the *Address Complement (AC)* CM. The AC CM specifies an address sequence: 000, **111**, 001, **110**, 010, **101**, 011, and **100** [100, 65]; each bold address is the 1's complement of the preceding address.

   The 2^i CM is yet another CM; typically used by the MOVI algorithm [100, 65]. It requires that the algorithm is repeated N times ($N =$ is the number of memory address bits) with an address increment/decrement value of 2^i; with $0 \leqslant i \leqslant N-1$.

Data Generation

The memory BIST has to provide the commonly used data backgrounds in industry [49]; it also has to generate the inverse of the used data background. They will be explained next.

1. *The Data Background (DB)*: is the data pattern which is actually in the cells of the memory cell array. The most common DBs used in industry are given next [49]:

   Solid (sDB): all 0s (i.e., 0000.../0000...) or all 1s.
   Checkerboard (bDB): 0101.../1010.../0101.../1010...
   Row Stripes (rDB): 0000.../1111.../0000.../1111...
   Column Stripes (cDB): 0101.../0101.../0101.../0101...

2. *The Data Value (DV)*: determines the *normal* or *inverse* data background pattern. For example, a '$w0$' means that the selected DB is applied; a '$w1$' means that the *inverse* of that DB is applied.

Table 6.2: Generic March Elements

B	GME#	GME Description	Alg.#
0	0	$\Updownarrow (wD)$	1-30
0	1	$\Updownarrow (rD)$	1,3-4,10-13
0	2	$\Updownarrow (rD, w\overline{D})$	2-3,9-10,16,30
0	3	$\Updownarrow (rD, w\overline{D}, r\overline{D})$	4-5,8,29
0	4	$\Updownarrow (rD, w\overline{D}, r\overline{D}, wD, rD, w\overline{D})$	6-8
0	5	$\Updownarrow (rD, w\overline{D}, wD)$	6,8
0	6	$\Updownarrow (rD, w\overline{D}, wD, w\overline{D})$	6-8
0	7	$\Updownarrow (rD, w\overline{D}, r\overline{D}, wD)$	7,9-10,14
0	8	$\Updownarrow (rD, w\overline{D}, wD, w\overline{D}, r\overline{D})$	11
0	9	$\Updownarrow (rD, rD, wD, rD, w\overline{D})$	12
0	10	$\Updownarrow (rD, wD, rD, rD, w\overline{D}, r\overline{D})$	13
0	11	$\Updownarrow (wD, rD, w\overline{D})$	15
0	12	$\Updownarrow_v (wD_v, \Box(r\overline{D}, rD_v), w\overline{D}_v)$	20
0	13	$\Updownarrow (rD, rD)$	14
0	14	Del50	30
0	15	Del100	8
1	0	$\Updownarrow (wD)$	1-30
1	1	$\Updownarrow (rD)$	1,3-4,10-13
1	2	$\Uparrow_v (wD_v, \Uparrow_{-v} (r\overline{D}, rD_v), w\overline{D}_v)$	17
1	3	$\Uparrow_v (wD_v, \Uparrow_{X-v} (r\overline{D}, rD_v), w\overline{D}_v)$	18-19
1	4	$\Uparrow (wD_v, \Uparrow_{BF} (r\overline{D}, rD_v), w\overline{D}_v)$	21
1	5	$\Uparrow_v (wD_v, \Uparrow_{-v} (r\overline{D}), rD_v, w\overline{D}_v)$	22
1	6	$\Uparrow_v (wD_v, \Uparrow_{X-v} (r\overline{D}), rD_v, w\overline{D}_v)$	23-24
1	7	$\Uparrow (rD, w\overline{D}^h, r\overline{D})$	25
1	8	$\Uparrow (rD, w\overline{D}, r\overline{D}^h, r\overline{D})$	26
1	9	$\nearrow (wD_v^h, \Uparrow_{R-v} (r\overline{D}), rD_v, \Uparrow_{C-v} (r\overline{D}), rD_v, w\overline{D}_v)$	27
1	10	$\nearrow (wD_v^h, \Uparrow_{C-v} (r\overline{D}), w\overline{D}_v)$	28
$x \in \{R, C\}$; Data Value $D \in \{0, 1\}$; $\overline{D}$ = inverse of D			

6.3 Generic March Element concept

Inspecting Table 6.1 reveals that many test algorithms use the same March element; they only differ in the address order and/or data value used by the test. Let's consider for example MATS+ and March C- below.

MATS+ [1]: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$
March C- [100]: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$

The first and second March elements of both tests are the same; the third March element of MATS+ is the same as that of March C- except that the address order is different. In addition, except the first and the last March elements, all March elements of March C- have the form of $\Updownarrow (rD, w\overline{D})$ where D can be either 0 or 1. Hence, MATS+ can be realized with two Generic March Elements (GMEs), namely $\Updownarrow (wD)$ and $\Updownarrow (rD, w\overline{D})$, and March C- can be realized with the same GMEs and the GME $\Updownarrow (rD)$. Note that a GME is a typical March element that only specifies the operations and their generic data values; it is further orthogonal to all other specifications like address order, counting method, etc.

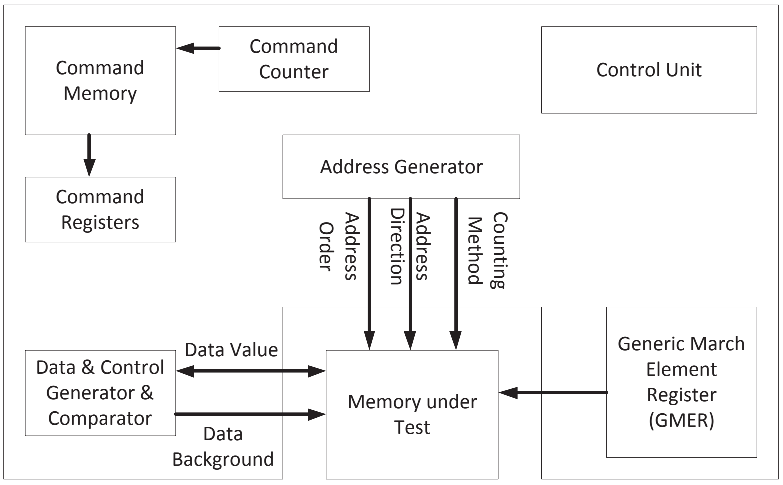


Figure 6.1: High level GME MBIST Engine architecture

Table 6.2 gives all GMEs required for the implementation of all memory test algorithms of Table 6.1. Moreover, these GMEs can realize many other algorithms that are not listed in Table 6.1 like IFA-6 [24, 25], March LRDD [103], March dADF [44], etc.

6.4 GME MBIST Engine architecture

The architecture of the GME MBIST Engine will be described in terms of its registers and its commands. Figure 6.1 shows a high-level block diagram with the 'Memory under Test', controlled by a set of orthogonal signals, which can be combined in any way. For example, the GME is orthogonal to any of the components of the Stress Combination (SC), for example the AO and the AD; this allows for the application of any GME with any SC.

Because of the open-ended architecture, extensions to customize this architecture are easy to make. Therefore, a subset, sufficiently large to illustrate the capabilities of the GME MBIST Engine, will be described.

Throughout the remainder of this section, the *basic* architecture will be covered together with a set of *extensions*, in terms of its registers and commands operating on these registers. The extensions illustrate the ease with which new features and capabilities can be added to the basic architecture. The commands and the registers are used to implement the tests, as shown in the examples of Section 6.5.

6.4.1 GME MBIST Register Set

The GME MBIST architecture supports registers which can be divided into two classes: the *Basic* registers and the *Extension* registers. The basic registers are part of the minimal GME MBIST Engine, while the *Extension* registers support some of the advanced features of the GME MBIST Engine. The register naming convention is such that the register name includes its size. For example, 'CC[5..0]' denotes the 6-bit Com-

Table 6.3: GME MBIST Engine registers

Register	B/E	Description
CM[63..0,3..0]	B	Command Memory
CC[5..0]	B	Command Counter
CR[7..0]	B	Command Register
AOR[0]	B	Address Order Register
DVR[0]	B	Data Value Register
CMADR[1..0]	B	Counting Method & Addr. Direction Reg.
DBR[1..0]	B	Data Background Register
GMER[3..0]	B	Generic March Element Register
BR[0]	E	Bank Register
REPR[5..0]	E	Re-execute Entry Point Register
RCNTR[3..0]	E	Re-execute CouNT Register
B/E = Basic or Extension		

mand Counter, its most significant bit is bit-5, its least significant bit is bit-0; 'AOR[0]' denotes a 1-bit Address Order Register. Table 6.3 summarizes the description of the register set of the GME MBIST Engine, which is described next. First the Basic registers are addressed, thereafter the Extension registers.

Basic registers

The basic registers are part of the minimal GME-MBIST engine; their presence is mandatory.

1. CM[63..0,3..0]: *Command Memory*
   Size depends on size of used test set; default: 64x4-bit nibbles. The CM contains the commands which specify the to-be-executed tests.
2. CC[5..0]: *Command Counter*
   Size depends on Command Memory (CM) size; default: 6 bits. The CC points to a location in the CM, which contains the to-be-executed command.
3. CR[7..0]: *Command Register*
   Contains the first 2 nibbles of the command.
4. MESR: *March Element Stress Register*
   The MESR specifies the to-be-applied Stress Combination (SC) which consists of 4 individual components, each described by its own register, as follows:

   - AOR[0]: *Address Order Register*
     This is a 1-bit register specifies the AO: $\Uparrow$ or $\Downarrow$.
   - DVR[0]: *Data Value Register*
     Specifies the Data Value (DV) used by the GME operations as follows:
     0: The GME operations assume the specified DB (see DBR)
     1: Use the inverted DB.
   - CMADR[1..0]: *Counting Method & Address Direction Register*
     This register specifies the combination of the to-be-used Counting Method (CM) and the Address Direction (AD), as follows:

Lr: Linear CM & Fast-row AD
Lc: Linear CM & Fast-column AD
AC: Address Complement CM; note that AD is not applicable to CM
2^i: i^{th} power of 2 CM; note that AD is not applicable to CM.

- DBR[1..0]: *Data Background Register*
  When the algorithm specifies a data value, such as for example in '$w0$', then the '0' value is interpreted as meaning the Data Background (DB) as specified in the DBR. This allows for the following DBs to be specified:
  sDB: solid DB
  bDB: checkerboard DB
  rDB: row stripes DB
  cDB: column stripes DB.

5. GMER[3..0]: *Generic March Element Register*
   The GMER is a 4-bit register which specifies the to-be-applied GME, within the current bank, as described in Table 6.2. Note that the GMER only contains a *number*, rather than a complete specification of a GME. The GME MBIST hardware uses the GME# to generate the appropriate sequence of read and write operations, together with their data values.

Extension registers

The extension registers are used to support additional features of the GME MBIST architecture; they are optional.

1. BR[0]: *Bank Register*
   The GMER allows for the specification of up to 16 GMEs. Some applications may have a need for a larger number of GMEs. In order to support this requirement, the set of GMEs is divided into *banks*, each containing up to 16 GMEs. The GME# specifies the GME in the bank, while the bank is specified via the BR. In this paper the size of the BR is 1-bit, because this already allows for up to 32 GMEs; however, depending on the application, it may have a different size.

2. REPR[5..0]: *Re-execute Entry Point Register*
   The REPR is used by the REP (REPeat) and the POWi commands, which allow a *Block of Commands (BoC)* to be re-executed a number of times. The starting address of this BoC is stored in the REPR, which has the same size as the CC (Command Counter).

3. RCNTR[3..0]: *Re-execute CouNT Register*
   The RCNT specifies the number of times a BoC has to be re-executed by the REP command. For the POWi command, the BoC is re-executed '$N-1$' times; N is the number of address bits. Note that the size of the RCNT register has to be the larger one of the following two values: 4, as needed by the REP command, and $\lceil log_2(N) \rceil$ for the POWi command.

Table 6.4: GME MBIST commands and command bit position assignments

		Bit positions in the command								
Command	BE	7	6	5	4	3	2	1	0	Xtra
INIT	B	Opc		AO	DV	CM&AD		DB		–
SGME	B	Opc		AO	DV	GME#				–
END	B	Opc								–
SAODV	E	Opc		AO	DV	–				–
SAODVE	E	Opc		AO	DV	CM&AD		Opc		GME#
SBR	E	Opc					B	Opc		–
SREP	E	Opc								–
REP	E	Opc		AO	DV	Opc				R
POWi	E	Opc		AO	DV	Opc				–

'BE' = Basic or Extension; 'Xtra'= extra nibbles
'–'= not applicable
'R' = RCNT#,CM&AD-DB$_{RCNT\#-1}$,...,CM&AD-DB$_2$,CM&AD-DB$_1$

6.4.2 GME MBIST Command Set

The architecture supports *commands* to control the operations of the GME MBIST Engine. These commands act like instructions on a traditional computer. However, the term *command* is used to avoid confusion with the host architecture of the GME MBIST Engine, which is likely to have the capability to execute *instructions*.

Two classes of commands can be recognized: Implicit and Explicit. *Implicit commands* are issued outside control of the user; typically as a side-effect of a system action or state (for example Power-On). The only implicit command for the GME MBIST Engine is the Power-On-Reset (POR) command. When the host system powers on, a hardwired POR command is given. This causes the following actions in the GME MBIST Engine:

- CC[5..0] $\leftarrow$ 0. Clear the Command Counter.
- BR[0] $\leftarrow$ 0. Clear the Bank Register.

All other commands are *Explicit*, which means that they have to be issued explicitly by the user. They are designed in such a way that they have a size (length) which is a multiple of 4 bits, also called a *nibble*. The length of the commands is *variable* in size, such that the Command Memory size can be minimized; the most frequent commands are encoded in the smallest number of nibbles. This reflects itself in the number of bits used for specifying the Opcode; which is also variable in length. Last, similar to the registers, the set of explicit commands can be divided into two classes: *Basic* and *Extension*. Table 6.4 summarizes the commands together with their bit assignments; the column 'Xtra' indicates the meaning of the extra nibbles for commands which require more than 2 nibbles. Note: the commands will be described in terms of their operations, which involve the basic and the extension registers. If a feature is not present, then the extension registers to support that features will not be present. However, the description of the commands assumes the presence of all features, in order to give a complete picture.

Basic Commands

The basic command set consists of *only three commands*; they are mandatory and allow for an absolute minimal implementation of the GME MBIST Engine (see Table 6.4).

1. INIT: *INITialize*
   The 2-nibble INIT command clears GMER. It establishes the DB values, the CM&AD, the DV and the AO for the application of the GME#0 specified in the GMER (which is '$\Updownarrow (wD)$'). Last, it establishes the Re-execute Entry Point by clearing the RCNTR and loading the start address of the to-be-re-executed Block of Commands in the REPR.

   Table 6.4 shows the INIT command. The entry 'B', in the column 'BE' indicates that it is a *Basic* command. The columns '7' through '0' show the bit assignment of the different fields of the command. For example, the Opc(ode) is assigned to bits 7 & 6, the AO field to bit 5, the Data Value (DV) field to bit 4, etc.

   INIT is a complex command, because it performs the following implicit and explicit operations, which are typical for the beginning of any test.

   - CC $\leftarrow$ CC + 1;
     Second nibble is fetched for CMAD and DB.
   - AOR $\leftarrow$ AO; DVR $\leftarrow$ DV; CMADR $\leftarrow$ CMAD; DBR $\leftarrow$ DB;
     AOR, DVR, CMADR and DBR are loaded.
   - GMER $\leftarrow$ 0; RCNTR $\leftarrow$ 0;
     GMER and RCNTR are reseted.
   - CC $\leftarrow$ CC + 1;
     CC is pointed to the next command.
   - REPR $\leftarrow$ CC;
     REPR is pointed to the next command after the INIT.
     Apply GME specified in GMER.

2. SGME: *Select GME*
   This command allows for the execution of a specified GME. It requires the specification of the GME#, the DV and the AO, as follows:

   - CC $\leftarrow$ CC + 1;
     Second nibble is fetched for GME#.
   - AOR $\leftarrow$ AO; DVR $\leftarrow$ DV; GMER $\leftarrow$ GME#;
     AOR, DVR and GMER are loaded.
   - CC $\leftarrow$ CC + 1;
     CC is pointed to the next command after the SGME command.
     Apply GME specified in GMER.

3. END: *END the test program*
   This command allows for the finalize the overall test program. The operation of the END command is as follows:

   - CC $\leftarrow$ CC + 1;
     Second nibble is fetched. CC is not incremented anymore for a next command, since the test is finalized.

Extension Commands

The extension commands allow for added capabilities of the basic GME MBIST Engine. These consist of extra functionality and/or a reduction of the required Command Memory size to represent tests (see Table 6.4).

1. SAODV: *Select AO and DV, for current GME*
   From inspecting Table 6.1, it can be seen that several algorithms have a sequence of MEs which only differ in the used AO and DV. For example: MATS+, March C- and PMOVI. This implies that only the *first* ME in a sequence has to be specified; the other MEs in the sequence only require the AO and DV to be specified. The operation of the SAODV command is as follows:

   - AOR ← AO; DVR ← DV;
     AOR and DVR are loaded.
   - CC ← CC + 1;
     CC is pointed to the next command after the SAODV command.
     Apply GME specified in GMER. Note: the SAODV command is 1 nibble in size.
     Example 2 in Section 6.5 shows its effectiveness.

2. SAODVE: *Select AO, DV, CM&AD and GME#*
   From inspecting Table 6.1 one can conclude that the selection of the 'CM&AD' applies to all MEs of the test. However, in some rare cases this does not hold. For example, the Philips 6n algorithm $\{_r\Uparrow(w0);\ _r\Uparrow(r0,w1);\ _c\Downarrow(r1,w0,r0)\}$ requires the use of the SAODVE command; this is because in addition to the GME#, the AO and the DV, the AD of the GME '$_c\Downarrow(r1,w0,r0)$' differs from the two preceding GMEs; see also Example 7 in Section 6.5.

   The suffix 'E' of the SAODVE command denotes an **E**xtension of the 'SAODV' command, in order to specify the GME# and CM&AD. This command is 3 nibbles in length; the third nibble specifies the GME#. The operation of SAODVE is as follows.

   - CC ← CC + 1;
     Second nibble is fetched for CMAD and second part of the opcode.
   - AOR ← AO; DVR ← DV; CMADR ← CMAD;
     AOR, DVR and CMDR are loaded.
   - CC ← CC + 1;
     Third nibble is fetched for GME#.
   - GMER ← GME#;
     GMER is loaded.
   - CC ← CC + 1;
     CC is pointed to the next command after the SAODVE command.
     Apply GME specified in GMER.

3. SBR: *Set Bank Register*
   The default size of the GMER[3..0] is 4 bits; such that one out of up to 16 GMEs can be specified. This choice is made to reduce the size of the Command Memory,

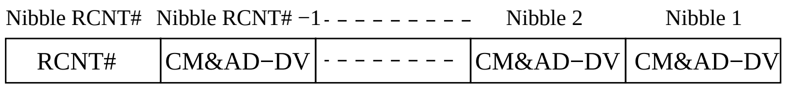


Figure 6.2: The 'Xtra' field of REP command

and because of the fact that commands have a size which is a multiple of a nibble. Therefore, the set of GMEs is divided into *banks* with a maximum of 16 GMEs. Note that if up to 64 GMEs have to be supported, then the BR will be 2 bits; i.e, BR[1..0], and the SBR has a 2-bit 'B' parameter. The Bank 'B' parameter of the SBR command specifies the bank as follows:

- CC ← CC + 1;
  Second nibble is fetched for B and second part of the opcode.
- BR ← B;
  BR is loaded.
- CC ← CC + 1;
  CC is pointed to the next command after the SBR command.
  The next command is fetched, it does not apply any GME.

4. SREP: *Set Re-execute Entry Point*
   Sometimes MEs of a set of tests are put together in a single long test. To save execution time, the state of the memory after a given test is used as the initial state for the following test. Then, if a *part* of the long test has to be re-executed, the SREP command is required to establish the Entry Point of that *part*. The operation of SREP is as follows:

   - CC ← CC + 1;
     Second nibble is fetched for second part of the opcode.
   - RCNTR ← 0;
     BR is reseted.
   - CC ← CC + 1;
     CC is pointed to the next command after the SREP command.
   - REPR ← CC;
     REPR is pointed to the next command after the SREP.
     The next command is fetched, it does not apply any GME.

5. REP: *REPeat block of commands*
   Industrial test sets usually contain tests which are a repeated application of an algorithm, whereby stress conditions such as the CM&AD are varied. The REPeat command supports this in a very efficient way. It allows for re-executing a *Block of Commands (BoC)*. The BoC starts at the location specified by the Re-execute Entry Point Register (REPR) and ends at the REP command. The BoC may consist of several algorithms and is repeated a number of times, as specified by the RCNT# field of the REP command; see Examples 5 and 8 in Section 6.5.

   The REP command consists of 2+RCNT# nibbles where the REP command will be repeated as the number of RCNT#-1 times ; see Table 6.4. The first 2 nibbles specify the initial AO and the DV. The 'Xtra' field of REP command with a length of RCNT# nibbles is shown in Figure 6.2. The first nibble in this field specifies the RCNT#. The additional RCNT#-1 nibbles specify the CM&AD and the DB values for the RCNT#-1 re-executions in *reverse order*; i.e., nibble 1 of Figure 6.2 specifies the CM&AD and the DB values for the first re-execution, nibble 2 for the second re-execution, and nibble RCNT#-1 for the last re-execution. Note that because the BoC has already been executed once before the REP command was encountered, the BoC will be re-executed RCNT#-1 times.

The operation of the REP command consists of the following two parts, determined by the contents of the register 'RCNTR': an *initialization* part and a *re-execution* part.

- CC ← CC + 1;
  Second nibble is fetched for second part of the opcode.

(a) Initialization part
If (RCNTR)=0 then:

- RCNTR ← RCNT# - 1;
  RCNTR is loaded.
- CC ← CC + RCNTR;
  The last nibble of the variable length command is fetched.
- AOR ← AO; DVR ← DV; CMADR ← CMAD#1; DBR ← DB#1;
  AOR, DVR, CMADR and DBR are loaded.
- GMER ← 0;
  GMER is reseted.
- CC ← REPR;
  CC points to the beginning of BoC.
  Re-execute the Block of Commands;

(b) Re-execution part
else if (RCNTR) $\neq$ 1 then:

- RCNTR ← RCNTR - 1;
  RCNTR is decremented by 1.
- CC ← CC + RCNTR; A nibble from the variable length command is fetched.
- AOR ← AO; DVR ← DV; CMADR ← CMAD#(RCNT#-RCNTR); DBR ← DB#(RCNT#-RCNTR);
  AOR, DVR, CMADR and DBR are loaded.
- GMER ← 0;
  GMER is reseted.
- CC ← REPR;
  CC points to the beginning of BoC.
  Re-execute the Block of Commands;

else:

- CC ← CC + RCNT#;
  The next command is fetched, it does not apply any GME. REP has been completed;

6. POWi: *Re-execute with POWer of 2^i addressing*
   The POWi command is a special case of the REP command: it re-executes a Block of Commands a fixed number of $N - 1$ times; N is the number of address bits, which is hardwired in the GME BIST Engine. During the i^{th} re-execution, i is used to generate the address increment/decrement value, which is 2^i. The RCNTR register is used to keep track of the current value of i. The operation of the POWi command is as follows:

   - CC ← CC + 1;
     Second nibble is fetched for second part of the opcode.

   (a) Initialization part
   If (RCNTR)=0 then:

   - RCNTR ← N - 1;
     RCNTR is loaded.
   - AOR ← AO; DVR ← DV;
     AOR and DVR are loaded.
   - GMER ← 0;
     GMER is reseted.
   - CC ← REPR;
     CC points to the beginning of BoC.
     Re-execute the Block of Commands;

   (b) Re-execution part
   else if (RCNTR $\neq$ 1) then:

   - RCNTR ← RCNTR - 1;
     RCNTR is decremented by 1.
   - AOR ← AO; DVR ← DV;
     AOR and DVR are loaded.
   - GMER ← 0;
     GMER is reseted.
   - CC ← REPR;
     CC points to the beginning of BoC.
     Re-execute the Block of Commands;

   else:

   - CC ← CC + 1;
     The next command is fetched, it does not apply any GME. POWi has been completed;

6.5 Memory test implementations

From the above it may be clear that only a few simple, memory efficient, commands are needed by the GME MBIST Engine. Their efficiency will be demonstrated with the code for following test programs:

1. March C-; it uses the minimal set of two commands.
2. March C-; it uses the extra command 'SAODV' to save command memory space.
3. GalPat; shows support of complex algorithms.
4. GalRow/GalCol; shows the use of the same GME with different CMAD.
5. Repeat (PMOVI & MATS+); shows the use of REP command.
6. MOVI; shows use of the POWi command.
7. Philip's $6n$ algorithm; shows the use of SAODVE command.
8. PMOVI & REPeat (MATS+); shows the use of SREP command.

March C-

Test description: $\{\Uparrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Uparrow (r0)\}$; applied with Lr and bDB with minimal command set as. This March C- test is implemented first by using only the *basic* commands, as follows:

Code:

- $\Uparrow (w0)$. INIT: AO= $\Uparrow$, DV=0, CMADR=Lr, DBR=bDB.
  Implicit application of GME#0.
- $\Uparrow (r0, w1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 2.
- $\Uparrow (r1, w0)$. SGME: AO= $\Uparrow$, DV= 1, GME#= 2.
- $\Downarrow (r0, w1)$. SGME: AO= $\Downarrow$, DV= 0, GME#= 2.
- $\Downarrow (r1, w0)$. SGME: AO= $\Downarrow$, DV= 1, GME#= 2.
- $\Uparrow (r0)\}$. SGME: AO= $\Uparrow$, DV= 0, GME#= 1.

A total of six commands are required, one per ME; they require: $6 * 2 = 12$ nibbles of command memory.

March C-

Test description: $\{\Uparrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Uparrow (r0)\}$. In this case, the same test as in Example 1 will be applied but then with bDB and Lr, and by using the *extended* command 'SAODV':

Code:

- $\Uparrow (w0)$. INIT: AO= $\Uparrow$, DV= 0, CMAD= Lr, DB= bDB.
  Implicit application of GME#0.
- $\Uparrow (r0, w1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 2.
- $\Uparrow (r1, w0)$. SAODV: AO= $\Uparrow$, DV= 1.
  Note: the GME# of the previous command applies.
- $\Downarrow (r0, w1)$. SAODV: AO= $\Downarrow$, DV= 0.
- $\Downarrow (r1, w0)$. SAODV: AO= $\Downarrow$, DV= 1.
- $\Uparrow (r0)\}$. SGME: AO= $\Uparrow$, DV= 0, GME#= 1.

Six commands require $3 * 2 + 3 * 1 = 9$ nibbles of command memory. Due to the use of 'SAODV' command, a **saving of 25%** is realized as compared with Example 1.

GalPat

Test description: $\{\Uparrow (w0); \Uparrow_v (w1_v, \Uparrow_{-v} (r0, r1_v), w0_v); \Uparrow (w1); \Uparrow_v (w0_v, \Uparrow_{-v} (r1, r0_v), w1_v)\}$; applied with sDB and Lr, as follows:

Code:

- SBR: B= 1.
  Note: Bank-1 of the set of GMEs has to be selected.
- $\Uparrow (w0)$. INIT: AO= $\Uparrow$, DV= 0, CMAD= Lr, DB= sDB.
  Implicit application of GME#0.
- $\Uparrow_v (w1_v, \Uparrow_{-v} (r0, r1_v), w0_v)$. SGME: AO= $\Uparrow$, DV= 1, GME#= 2.
- $\Uparrow (w1)$. SGME: AO: $\Uparrow$, DV: 1, GME#: 0.
- $\Uparrow_v (w0_v, \Uparrow_{-v} (r1, r0_v), w1_v)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 2.

Five commands require $5 * 2 = 10$ nibbles of command memory.

GalRow/GalCol

Test description: $\{\Uparrow (w0); \Uparrow_v (w1_v, \Uparrow_{X-v} (r0, r1_v), w0_v); \Uparrow (w1); \Uparrow_v (w0_v, \Uparrow_{X-v} (r1, r0_v), w1_v)\}$; applied with sDB and Lc, as follows:

Code:

- SBR: B= 1.
  Bank 1 is selected.
- $\Uparrow (w0)$. INIT: AO= $\Uparrow$, DV= 0, CMAD= Lc/Lr, DB= sDB.
  Implicit application of GME#0. Lc for GalRow, Lr for GalCol.
- $\Uparrow_v (w1_v, \Uparrow_{X-v} (r0, r1_v), w0_v)$. SGME: AO= $\Uparrow$, DV= 1, GME#= 3.
- $\Uparrow (w1)$. SGME: AO= $\Uparrow$, DV= 1, GME#= 0.
- $\Uparrow_v (w0_v, \Uparrow_{X-v} (r1, r0_v), w1_v)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 3.

The required command memory is $5 * 2 = 10$ nibbles.

REPeat(PMOVI & MATS+)

Test description:
PMOVI: $\{\Downarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0)\}$
MATS+: $\{\Uparrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$.

This example demonstrates the use of the *REP command* applied to two tests: PMOVI & MATS+. The first time they are executed using Lr (Linear addressing & Fast-row), together with sDB (solid DB). Then they will be re-executed for Lr & bDB, Lr & rDB, Lr & cDB, and for Lc & sDB, Lc & bDB, Lc & rDB and Lc & cDB. This means that, after the initial execution, they are re-executed 7 times. The following set of commands will implement the above:

Code:
PMOVI

- $\Downarrow (w0)$. INIT: AO= $\Downarrow$, DV= 0, CMAD= Lr, DB= sDB.
  Apply implicit GME#0. Note that RCNTR will be implicitly cleared here.
- $\Uparrow (r0, w1, r1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 3.
- $\Uparrow (r1, w0, r0)$. SAODV: AO= $\Uparrow$, DV= 1.
- $\Downarrow (r0, w1, r1)$. SAODV: AO= $\Downarrow$, DV= 0.
- $\Downarrow (r1, w0, r0)$. SAODV: AO= $\Downarrow$, DV= 1.

MATS+

- $\Uparrow (w0)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 0.
- $\Uparrow (r0, w1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 2.
- $\Downarrow (r1, w0)$. SAODV: AO= $\Downarrow$, DV= 1.
- REP: AO= $\Uparrow$, DV= 0, RCNT#= 8, CMAD-DB#7= Lc-cDB, CMAD-DB#6= Lc-rDB, CMAD-DB#5= Lc-bDB, CMAD-DB#4= Lc-sDB, CMAD-DB#3= Lr-cDB, CMAD-DB#2= Lr-rDB, CMAD-DB#1= Lr-bDB.

Nine commands require $4 * 2 + 4 * 1 + 1 * 10 = 22$ nibbles of command memory.

MOVI

Test description: $\{\Downarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0)\}$; apply 2^i CM with Lr and bDB. This is a special algorithm, because MOVI is a repeated application of the PMOVI algorithm, under control of the special 2^i CM. The following set of commands will implement the above:

Code:

- $\Downarrow (w0)$. INIT: AO= $\Downarrow$, DV= 0, CMAD= Lr, DB= bDB.
  Apply implicit GME#0.
- $\Uparrow (r0, w1, r1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 3.
- $\Uparrow (r1, w0, r0)$. SAODV: AO= $\Uparrow$, DV= 1.
- $\Downarrow (r0, w1, r1)$. SAODV: AO= $\Downarrow$, DV= 0.
- $\Downarrow (r1, w0, r0)$. SAODV: AO= $\Downarrow$, DV= 1.
- POWi: AO: $\Uparrow$, DV: 0.
  This will repeat the above set of commands $N - 1$ times with address increment /decrements of 2^i.

Six commands require $3 * 2 + 3 * 1 = 9$ nibbles of command memory.

Philip's $6n$ algorithm

Test description: $\{_r\Uparrow(w0);_r\Uparrow(r0,w1);_c\Downarrow(r1,w0,r0)\}$; with sDB. Note that the AD of the first two MEs is Fast-row, while the AD of the last ME is Fast-column. The implementation uses the command 'SAODVE' as follows:

Code:

- $_r\Uparrow(w0)$. INIT: AO= $\Uparrow$, DV= 0, CMAD= Lr, DB= sDB.
  Apply implicit GME#0.
- $_r\Uparrow(r0,w1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 2.
- $_c\Downarrow(r1,w0,r0)$. SAODVE: AO= $\Downarrow$, DV= 1, CMAD= Lc, GME#= 3.

Three commands require $2*2+1*3=7$ nibbles of command memory.

PMOVI & REPeat (MATS+)

Test description: Two tests are joined together to form a single long test. The final memory state of PMOVI is used as the initial state by MATS+; that way saving the initializing operation '$\Uparrow(w0)$' for MATS+. The long test is applied with Lc & sDB; then the MATS+ part is re-executed with Lr-bDB. This results in the following program, using the SREP (Set Re-execute Entry Point) command:

Code:

- $\Downarrow(w0)$. **Begin of PMOVI**. INIT: AO= $\Downarrow$, DV= 0, CMAD= Lc, DB= sDB.
  Apply implicit GME#0.
- $\Uparrow(r0,w1,r1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 3.
- $\Uparrow(r1,w0,r0)$. SAODV: AO= $\Uparrow$, DV= 1.
- $\Downarrow(r0,w1,r1)$. SAODV: AO= $\Downarrow$, DV= 0.
- $\Downarrow(r1,w0,r0)$. **End of PMOVI**. SAODV: AO= $\Downarrow$, DV= 1.
- SREP: **Begin of MATS+**. This establishes the re-execution entry point!
- $\Uparrow(r0,w1)$. SGME: AO= $\Uparrow$, DV= 0, GME#= 2.
- $\Downarrow(r1,w0)$. SAODV: AO= $\Downarrow$, DV= 1.
- REP: AO= $\Uparrow$, DV= 0, RCNT#= 2. CMAD#1= Lr, DB= bDB.

The command memory will require $4*2+4*1+1*4=16$ nibbles.

GME MBIST architecture implementation 7

This chapter covers the implementation and evaluation of the GME MBIST Engine. Section 7.1 gives the functional model of the MBIST architecture. It consists of two major parts; the Memory BIST Processor and the Memory Wrapper. Section 7.2 discusses the Memory BIST Processor. Section 7.3 addresses the Memory Wrapper. Section 7.4 investigates the optimization of address generator. Section 7.5 presents the experimental results of the GME MBIST Engine. Section 7.6 provides a comparison with the state-of-the-art. Section 7.7 concludes with the advantages of GME MBIST over previous state-of-the-art implementations.

7.1 GME MBIST Engine

The GME MBIST Engine is a built-in self-test hardware that performs the user defined memory tests. Figure 7.1 shows a high level block diagram of GME MBIST Engine consisting of two main sub-blocks: GME MBIST Processor and GME MBIST Wrapper. GME MBIST Processor is responsible for the management of test flow, whereas GME MBIST Wrapper is responsible for the implementation of the test signal generation (i.e., address, data and control signal generation) and the analysis of the memory output response.

GME MBIST has two operation modes: normal and test modes. In the normal mode, GME MBIST delivers the normal mode signals (e.g., memory address, data, control signals) incoming from the system to the memory without any interference. In addition, the test commands are loaded during the normal mode. After commands are loaded, test flow is initiated and GME MBIST enters the test mode. In test mode, the normal mode access is bypassed; the test mode signals generated by GME MBIST are delivered to the memory. In case of a memory fault detection, the address and the readout data of the faulty cell are delivered out. When the overall test flow ends, GME MBIST returns to the normal mode and keeps waiting in an idle state for a new test start.

7.2 GME MBIST Processor

GME MBIST Processor is responsible of the test flow and consists of three sub-blocks: Command Memory (CM), Controller and Registers as shown in Figure 7.2. CM stores the test commands. Controller selects the to-be fetched command from CM; decodes it and configures the Registers. In addition, test flow is initiated and finalized by the Controller. Registers store the configuration parameters for the memory tests.

GME MBIST Processor sends the contents of the Registers to the GME MBIST Wrapper. Afterwards, it initiates the test flow by activating the GME MBIST Wrapper.

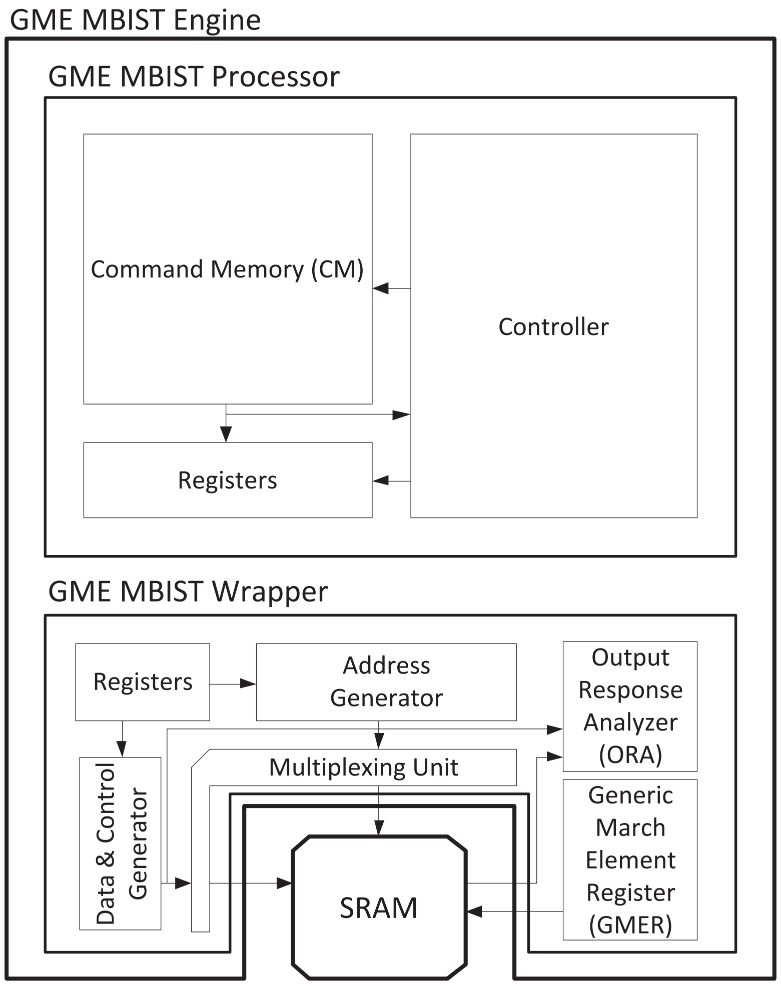


Figure 7.1: High Level architecture of GME MBIST Engine

During a GME is being performed to the memory, GME MBIST Processor waits in an idle state. When performing a GME ends, GME MBIST Processor continues with the next command.

7.2.1 Command Memory

CM is the sub-block where the test commands are stored. By default, its size is 64 x 4-bit nibbles as shown in Figure 7.3. The to-be fetched nibble is determined by the Controller. Table 6.4 shows the physical mapping of the commands. Notice that commands and opcodes are variable-length for an efficient usage of CM. The list below gives the encodings of the test parameters.

- AO: '0' for Up, '1' for Down,
- DV: '0' for Normal, '1' for Inverse,
- CM&AD: "00" for Lr, "01" for Lc, "10" for AC, "11" for 2^i,

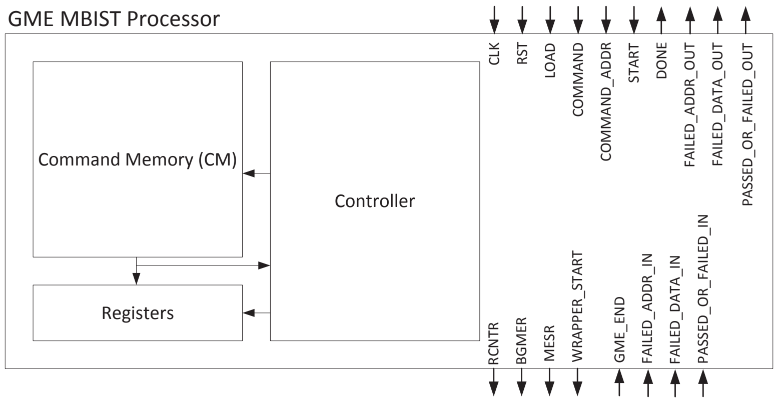


Figure 7.2: GME MBIST Processor architecture

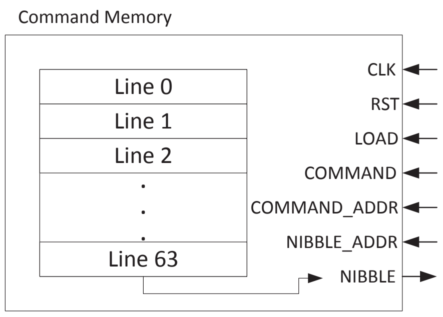


Figure 7.3: Command Memory architecture

- DB: ”00” for sDB, ”01” for bDB, ”10” for rDB, ”11” for cDB,
- B: ’0’ for Bank 0, ’1’ for Bank 1,
- GME#: from ”0000” (0) to ”1111” (15),
- RCNT#: from ”0010” (2) to ”1111” (15); since, ”REP” command repeats a BoC as RCNT#-1 times, RCNT# can be minimum 2, whereas due to the 4-bit binary coding, it can be maximum 15.

Figure 7.4 shows the binary coding of the GME MBIST commands. Less frequently used commands have the longest opcodes, while the most frequently used ones have the shortest. ”INIT”, ”SGME” and ”SAODV” commands are encoded as: ”00”, ”01” and ”10”, respectively. Then the 1^{st} ”Opc” area becomes ”11”, and 2^{nd} ”Opc” area follows as: ”00”, ”01” and ”10” for ”SAODVE”, ”SBR” and ”SREP” commands, respectively. Then the 2^{nd} ”Opc” area becomes ”11”, and 3^{rd} ”Opc” area follows as: ”00”, ”01” and ”10” for ”REP”, ”POWi” and ”END” commands, respectively. This coding allows further extensions on command set.

7.2.2 Controller

Controller directs the test flow. It controls the Command Counter (CC) which contains the address of the to-be fetched nibble from the CM. Figure 7.5 shows that it consists of a

	Opc							
	7	6	5	4	3	2	1	0
INIT	0	0						
SGME	0	1						
SAODV	1	0						
SAODVE	1	1					0	0
SBR	1	1	X	X	X		0	1
SREP	1	1	X	X	X	X	1	0
REP	1	1			0	0	1	1
POWi	1	1			0	1	1	1
END	1	1	X	X	1	0	1	1

Figure 7.4: Binary coding of the GME MBIST commands

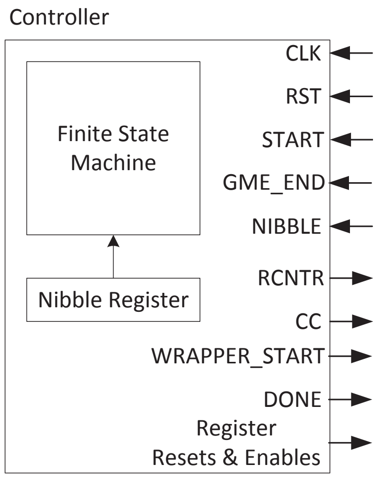


Figure 7.5: Controller architecture

register and a finite state machine. The register stores the fetched nibble. FSM performs the operations (e.g., reseting/loading of registers, loading CC) for each command as explained in Chapter 6. For control state details, please see Appendix A.

7.2.3 Registers

Registers store the information of the GME and algorithm stresses that will be performed to the memory. Figure 7.6 shows the 2 registers: MESR (i.e., combination of the AOR, DVR, CMADR, DBR) and BGMER (i.e., combination of the BR, GMER). Those registers are reseted and/or loaded by the control signals from the Controller.

7.3 GME MBIST Memory Wrapper

GME MBIST Wrapper generates the test signals (i.e., address, data and control) and analyzes the memory output response. It is closely located to the Memory-under-Test to enable at-speed testing. It consists of five sub-blocks: Registers, Address Generator,

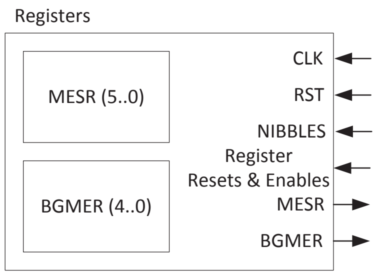


Figure 7.6: Registers of the GME MBIST Processor architecture

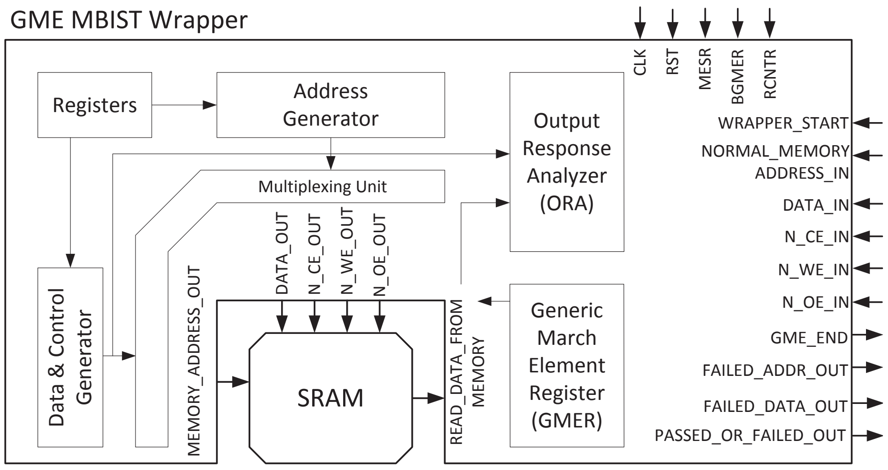


Figure 7.7: GME MBIST Wrapper architecture

Data & Control Generator, Multiplexing Unit and Output Response Analyzer (ORA) as shown in Figure 7.7. The registers stores the information of GME and algorithm stresses; the Address Generator generates the to-be applied test address to the memory; the Data & Control Generator generates the to-be applied test data and control signals to the memory; Multiplexing Unit selects between the normal or test mode signals; the Output Response Analyzer checks the validity of the memory response.

GME MBIST Wrapper is activated by the GME MBIST Processor after the Registers are loaded. Specified GME#, address, data and control signals are generated in orthogonal to each other. Multiplexing Unit bypasses the normal mode signals, and it delivers the test mode signals to the memory. Simultaneously, the reference signals (i.e., data and control) are sent to the ORA. Afterwards, ORA compares the reference data and the response data received from the memory. In case of a faulty behavior, address and data of the faulty cell are reported to the user back.

7.3.1 Registers

The registers stores the information of GME and algorithm stresses. Figure 7.8 shows that it consists of the MESR and BGMER registers. They are loaded before the test signal generation.

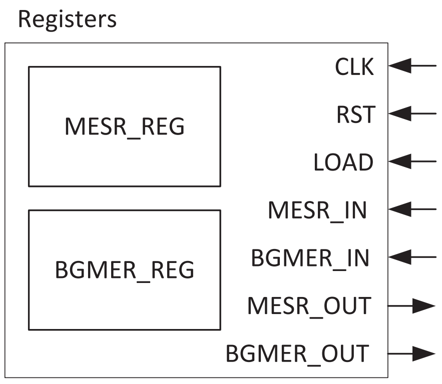


Figure 7.8: Registers of the GME MBIST Wrapper architecture

7.3.2 Address Generator

This module generates the test addresses. Figure 7.9 shows that it consists of several counters for different memory tests. To minimize the module area overhead, an up-only counter is chosen. Different counting methods, address orders and directions are achieved by the Barrel Shifter and MUX Unit at the output of the Up-only Counter. Barrel Shifter is used for the 2^i addressing, whereas MUX Unit is for the Lr, Lc and AC addressing schemes. For the two-level nested loops of the non-linear algorithms (e.g., GalPat, Walking 1/0), a register is added. Address Generator stores the base cell address at the register before it enters to a nested loop. When a nested loop ends, the base address is loaded back to the Up-only Counter.

For the Butterfly algorithm, one row adder and one column adder are optionally added. They are also used by the Gal9R test. Similarly for the Hammer and Delay tests, there are two extra counter, too. For control state details and VHDL code examples, please see Appendix B and C, respectively.

Since the Address Generator occupies a large area and dominates the GME MBIST hardware, further study is focused on the optimization of the Address Generator. Techniques and results are presented in Section 7.4.

7.3.3 Data & Control Generator

This module generates the test data and control signals. Figure 7.10 shows the data and control generation sub-blocks.

Data background pattern is simply generated by AND/XORing the row/column addresses with each other. For the solid data background, DV value determines the data value independent from the current address; for the checkerboard, the least significant bits of the row and column addresses (row_addr(0) and column_addr(0)) and DV are XORed; for the row stripe, the least significant bit of the row address and DV are XORed; for the column stripe, the least significant bit of the column address and DV are XORed.

Control signals are simply generated depending on the two memory operations: read and write. In case of a read operation, the chip enable and the output enable are

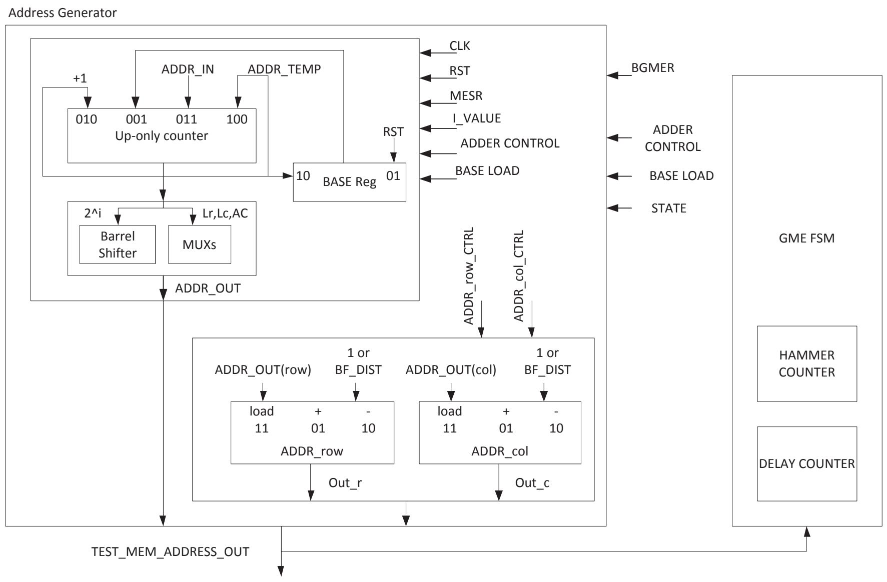


Figure 7.9: Address Generator architecture

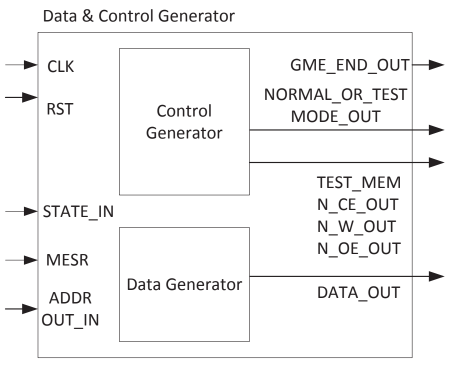


Figure 7.10: Data & Control Generator architecture

LOW, whereas write enable is HIGH; for a write operation, the chip enable and the write enable are LOW, whereas the output enable is HIGH, since they are low asserted signals. For control state details and VHDL code examples, please see Appendix B and C, respectively.

7.3.4 Output Response Analyzer

Output Response Analyzer detects the faults in the Memory-under-Test. It checks the validity of the memory output response by comparing with the reference data. Figure 7.11 shows that ORA consists of a few registers and a comparator. Since the memory response is received only for the read operation, comparison is performed only for the read operation.

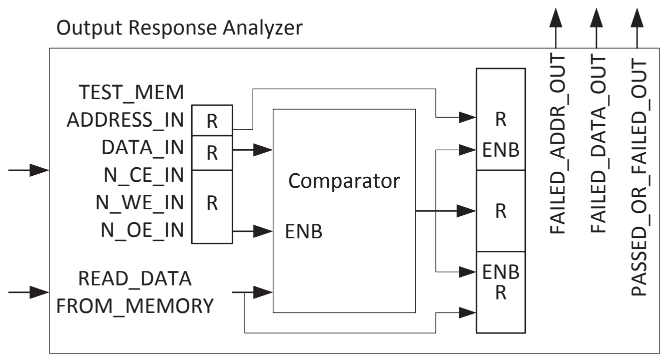


Figure 7.11: Output Response Analyzer architecture

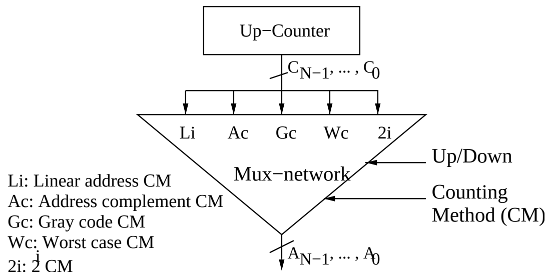


Figure 7.12: Up-counter based address generator

7.4 Optimization of the Address Generator

This section presents a novel concept to optimize the area and power of the Address Generator. As shown in Figure 7.12 only an Up-counter is used to create the Linear Up-address sequence; and the multiplexer network builds the required Up/Down Counting Method.

Rest of the section follows as: Section 7.4.1 explains the concept for the Linear (Li) and Address complement (Ac) Address Generators (AddrGens). Section 7.4.2 presents the area and power analysis. Section 7.4.3 discusses the Gray code (Gc) and the Worst-case gate delay (Wc) AddrGens. Section 7.4.4 presents the 2^i (2i) AddrGen. Section 7.4.5 focuses on the Pseudo-random (Pr) AddrGen. Section 7.4.6 continues with the Next (Ne) AddrGen. Finally, Section 7.4.7 summarizes the results.

7.4.1 Linear and Address complement AddrGens

Table 7.1 show the Address Sequences (AS) of different CMs for a 4-bit address. Column Li corresponds to the Linear AS, while Ac shows the Address complement AS, etc.

LiUd: Linear AddrGen based on Up-down counter

Figure 7.13(a) shows a J-K flip-flop based Up-down counter (LiUd). Address direction ('$\Uparrow$' or '$\Downarrow$') is controlled by the 'U/D' (Up/Down) which is ANDed and ORed with the previous J-K stage outputs to determine the inputs of the next stage.

Table 7.1: Address Counting Methods(CMs)

Step	**Li**	**Ac**	**Gc**	$2^i = 4$	Pr	Wc
0	0000	0000	0000	0**0**00	0000	-
1	0001	**1111**	0001	0**1**00	0001	000**1**
2	0010	0001	0011	1**0**00	0011	0000
3	0011	**1110**	0010	1**1**00	0111	000**1**
4	0100	0010	0110	0**0**01	1111	-
5	0101	**1101**	0111	0**1**01	1110	00**1**0
6	0110	0011	0101	1**0**01	1101	0000
7	0111	**1100**	0100	1**1**01	1010	00**1**0
8	1000	0100	1100	0**0**10	0101	-
9	1001	**1011**	1101	0**1**10	1011	0**1**00
10	1010	0101	1111	1**0**10	0110	0000
11	1011	**1010**	1110	1**1**10	1100	0**1**00
12	1100	0110	1010	0**0**11	1001	-
13	1101	**1001**	1011	0**1**11	0010	**1**000
14	1110	0111	1001	1**0**11	0100	0000
15	1111	**1000**	1000	1**1**11	1000	**1**000

Note: Li= Linear; Ac= Address complement; Gc= Gray code; Pr= Pseudo-random; Wc= Worst-case gate delay

LiUo: Linear AddrGen based on Up-only counter
Figure 7.13(b) shows an Up-only counter based Up-down counter (LiUo). Output of the counter is multiplexed depending on the U/D to obtain the ⇑ or the ⇓ AS.

Ac: Address complement AddrGen
Figure 7.13(c) shows an Up-only counter based Ac implementation. The 'U/D' multiplexes the least significant counter bit $C0$ to obtain the most-significant address bit $A3$. The Q output of $C0$ multiplexes the rest.

LiAc: Combined LiUo & Ac AddrGen
Figure 7.13(d) shows an Up-only counter based combined Li&Ac implementation. The CTLR1 and CTRL2 control signals consist of 2 bits. They multiplex the counter output as shown in the bottom-left of the figure; e.g., CTRL1=2 means LiU, etc.

7.4.2 Area and power analysis of Li and Ac AddrGens

Synthesis was performed with the Synopsys Design Compiler [58], using the Faraday UMC 90 nm Standard Process library [22]. Table 7.2 presents the normalized area, *in terms of standard 2-input NAND gates*, for the the LiUd, the LiUo, the Ac, and the combined LiAc. Results are taken at three frequencies (555, 833, 1111 MHz) for increasing counter size (N=8, 12, 16, 20 and 24 bits).

The area increases for higher N and frequency values. '$\triangle$Area Freq in %' corresponds to the area increase from the frequency of 555 to 1111 MHz. Increasing the frequency does not increase the number of gates, the design is kept same; however, in order to meet the required clock frequency, certain gates are resized to get more drive strength; hence, more area overhead. As shown in the table the LiUd has the largest area increase for increasing N, which is between 30.7 and 45.3%.

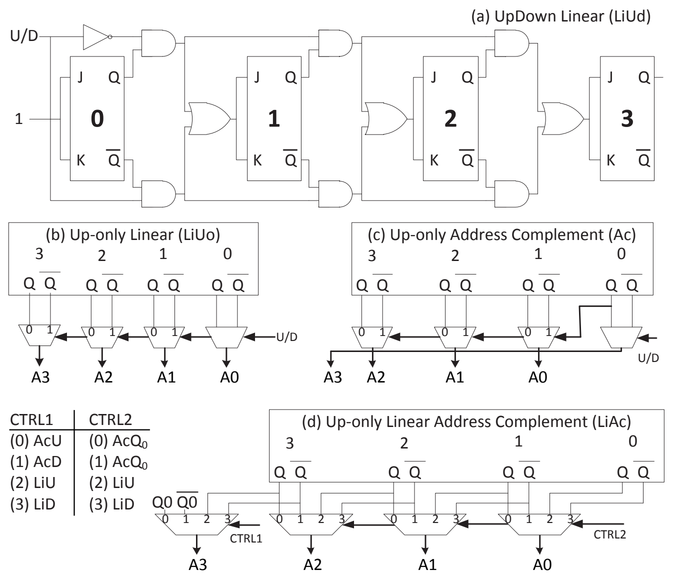


Figure 7.13: Li & Ac AddrGens

Table 7.2: Area metrics of Li & Ac AddrGens

Cntr	**Freq** in MHz	N				
		8	12	16	20	24
LiUd	555	123	186	262	344	426
LiUd	833	135	219	305	401	500
LiUd	1111	179	265	360	455	556
△Area Freq in %		45.3	41.9	37.2	32.3	30.7
LiUo	555	107	170	230	286	352
LiUo	833	110	172	234	297	365
LiUo	1111	116	191	274	355	435
△Area Freq in %		8.4	12.6	19.4	24.0	23.6
△Area LiUd-Uo in %		35.2	27.9	23.8	22.0	21.8
Ac	555	108	168	227	289	351
Ac	833	112	171	230	299	362
Ac	1111	114	192	273	353	435
△Area Freq in %		5.3	13.8	20.2	22.3	24.1
LiAc	555	122	182	252	325	388
LiAc	833	134	202	269	341	414
LiAc	1111	139	227	313	396	486
△Area Freq in %		14.1	24.8	24.3	22.0	25.1

Moreover, the table reveals that the LiUd requires the largest area for increasing frequencies; e.g., LiUd consumes 21.8 to 35.2% more than LiUo; this is given in row '△Area LiUd-Uo in %'.

Figure 7.14 shows the power analysis of the LiUd and the LiUo at three frequencies; the LiUd is worse, especially for higher frequencies, by 13 to 23%. The power increases non-linearly with the frequency, since a higher frequency also demands for a larger circuit area; see Table 7.2. Considering the advantages the LiUo counter has over the LiUd counter, the latter will not be considered any more from this point on.

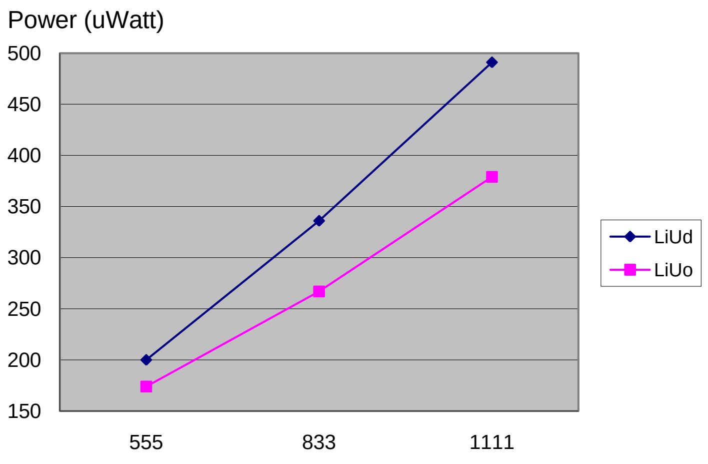


Figure 7.14: Power in μWatts for LiUd & LiUo

7.4.3 Minimizing the Gray code and Worst-case gate delay AddrGens

This subsection shows how the Up-only counter is used to generate optimized AddrGens for Gray code (Gc) and the Worst-case gate delay (Wc) CMs.

Gc: Gray code AddrGen

The column 'Gc' of Table 7.1 shows a 4-bit address sequence for the Gc CM. This sequence can be derived from the Linear sequence, as follows: bit$_0$ of the Gc address can be derived from bit$_0$ of the Linear address by inverting it when bit$_1$ of the linear address is '1'. This is shown in Figure 7.15(a): the mux of bit$_0$ is controlled by the signal 'Q1'. Similar reasoning applies to bit$_1$ and bit$_2$. The mux of bit$_3$ is controlled by the Up/Down signal, which means that in case of the $\Uparrow$ address sequence, the '0' input of the mux will select $Q3$ to generate $O3$; see Table 7.1. Based on the above, the implementation of Gc AddrGen is done in a simple and easy way by using linear Up-only counter.

Wc: Worst-case gate delay AddrGen

The Worst-Case Gate Delay (WCGD) algorithm [108] has been designed as a more efficient replacement of the MOVI algorithm [23]. It has the property that for each of the 2^N victim addresses ($v_{addr}es$), the following N address-triplets are generated: $v_{addr} \oplus 2^j, v_{addr}, v_{addr} \oplus 2^j$; for $0 \leqslant j \leqslant N-1$. The column 'Wc' of Table 7.1 sketches part of a 4-bit Wc address sequence; i.e., for $v_{addr} = 0000$. For every v_{addr} of the register ($Q3, Q2, Q1$, and $Q0$), any one of the 4 address bits has to be inverted. This is accomplished in Figure 7.15(b) by selecting the Qj or the $\overline{Qj}$ output, under the control of the corresponding mux with control input '2ˆ j'. For example, for bit$_2$ the mux control input is labeled '2ˆ j' and (2). Note that of the 4 mux control inputs *only one* is active, such that only *one address bit* is inverted.

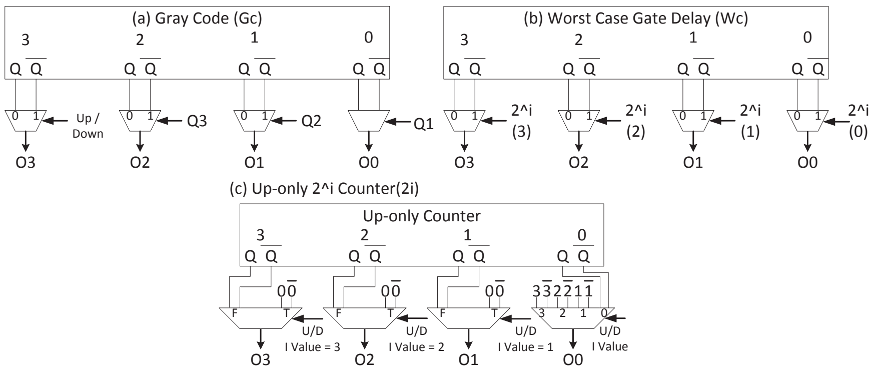


Figure 7.15: Gc, Wc, and 2^i AddrGens

7.4.4 2^i AddrGen

The 2^i CM is important for the MOVI algorithm [65, 100], which is used throughout the industry. Table 7.3 will be used to explain the AS. The sub-table 'Regular 2i CM' lists the 'regular' 2^i CM. In the column '0' stands for '$i = 0$': aids of $2^0 = 1$ are used; see column in **bold** font. In the next column aids of $2^1 = 2$ are used, etc. A barrel shifter with N muxes, each with N inputs, could be used to transform the Li AS into the 'Regular' 2^i AS. However, this requires a total of: $N * N = N^2$ inputs.

The second sub-table of Table 7.3, 'Economical 2i CM', shows the operation of the economical solution. The AS in the column '$i = 0$' is identical to the regular AS. For all other values of i the muxes interchange col_i with col_0; see **bold** columns. Therefore, the mux for bit_0 requires N inputs, while the other muxes only require 2 inputs. Implementation of the economical solution is shown in Figure 7.15(c); the required number of mux inputs are reduced to: $2 * (N - 1) + N = 3N - 2$.

Table 7.3: Ways of 2^i addressing

	Regular 2^i CM				Economical 2^i CM			
#	0	1	2	3	0	1	2	3
0	000**0**	00**0**0	0**0**00	**0**000	0000	00**00**	0**0**0**0**	**0**00**0**
1	000**1**	00**1**0	0**1**00	**1**000	0001	00**10**	0**1**0**0**	**1**00**0**
2	001**0**	01**0**0	1**0**00	**0**001	0010	00**01**	0**0**1**0**	**0**01**0**
3	001**1**	01**1**0	1**1**00	**1**001	0011	00**11**	0**1**1**0**	**1**01**0**
4	010**0**	10**0**0	0**0**01	**0**010	0100	01**00**	0**0**0**1**	**0**10**0**
5	010**1**	10**1**0	0**1**01	**1**010	0101	01**10**	0**1**0**1**	**1**10**0**
6	011**0**	11**0**0	1**0**01	**0**011	0110	01**01**	0**0**1**1**	**0**11**0**
7	011**1**	11**1**0	1**1**01	**1**011	0111	01**11**	0**1**1**1**	**1**11**0**
8	100**0**	00**0**1	0**0**10	**0**100	1000	10**00**	1**0**0**0**	**0**00**1**
9	100**1**	00**1**1	0**1**10	**1**100	1001	10**10**	1**1**0**0**	**1**00**1**
10	101**0**	01**0**1	1010	**0**101	1010	10**01**	1**0**1**0**	**0**01**1**
11	101**1**	01**1**1	1**1**10	**1**101	1011	10**11**	1**1**1**0**	**1**01**1**
12	110**0**	10**0**1	0**0**11	**0**110	1100	11**00**	1**0**0**1**	**0**10**1**
13	110**1**	10**1**1	0**1**11	**1**110	1101	11**10**	1**1**0**1**	**1**10**1**
14	111**0**	11**0**1	1**0**11	**0**111	1110	11**01**	1**0**1**1**	**0**11**1**
15	111**1**	11**1**1	1**1**11	**1**111	1111	11**11**	1**1**1**1**	**1**11**1**

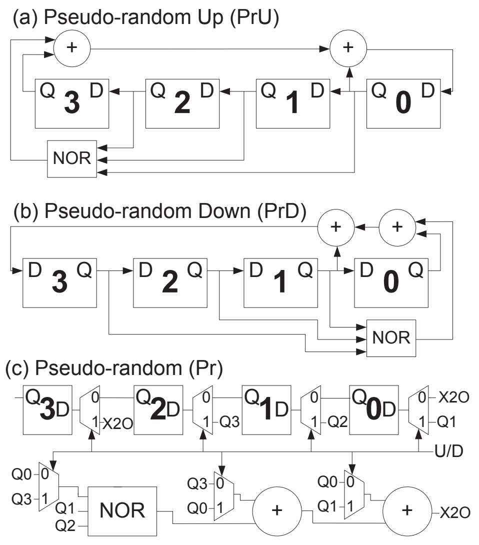


Figure 7.16: Pr AddrGen

7.4.5 Pseudo-random AddrGen

The implementation of Pr AddrGen requires a Linear Feedback Shift Register (LFSR); see Figure 7.16(a). It can generate the Address Sequence (AS) of the column 'PR' of Table 7.1, which we will denote the $\Uparrow$ AS. For this the LFSR uses the primitive polynomial 'G(x)': $G(x) = x^4 + x + 1$, such that the maximum-length sequence will be generated [100]. This polynomial is implemented by XORing the outputs of bit$_3$ and bit$_0$ and feeding it to the input of LFSR. The LFSR has to shift to the left; i.e., towards the most significant address bit. The NOR gate allows for the generation of the all-0 address; when the state of the LFSR is 1000 or 0111, it inserts a '1' into the XOR network. That way it can get out of the state '0000'.

When the $\Downarrow$ AS has to be generated, the LFSR has to shift towards the least-significant bit, while the XOR network has to implement the reverse polynomial $G^*(x)$, which satisfies the equation: $G^*(x) = x^g * G(1/x)$; g is the *degree* of the polynomial [100]. The reverse polynomial $G^*(x) = x^4 * (1/x^4 + 1/x + 1) = x^4 + x + 1$ is implemented in the 'Pseudo-random Down (PrD)'; see Figure 7.16(b).

Figure 7.16(c) shows the 4-bit Pr AG, which can generate the $\Uparrow$ and the $\Downarrow$ AS; it is a combination of Figure 7.16(a) and (b). The left and right shift capability is supported by the muxes located between the LFSR cells; which are controlled by the Up/Down signal.

7.4.6 Next-address AddrGen

Figure 7.17 shows the Next(Ne) AddrGen. The implementation is based on the idea that the increment function of the Up-only counter can be separated from the Register function. This results in two separate units: the 'Register' and the '+1 increment logic' as shown in Figure 7.15(b). To generate the $\Uparrow$ and $\Downarrow$ sequences, the mux in the figure can select the Register outputs, which represent the 'Normal Sequence', via mux inputs

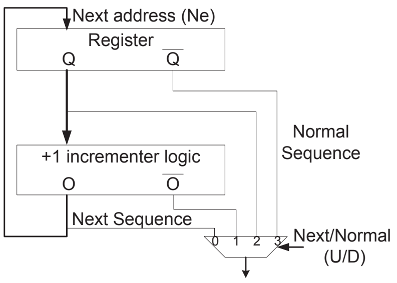


Figure 7.17: Ne AddrGen

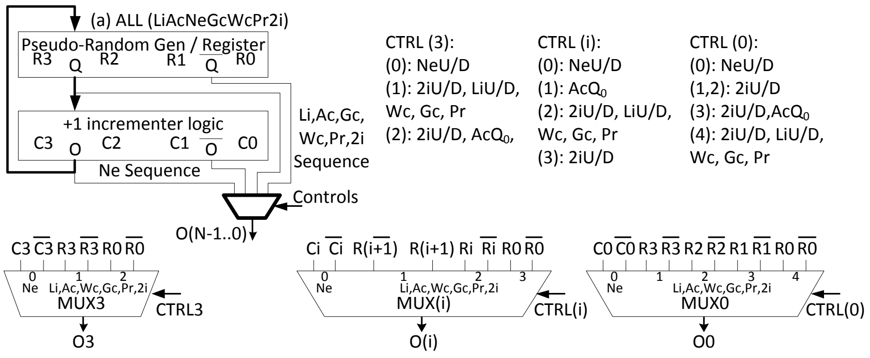

(a) ALL (LiAcNeGcWcPr2i)
Pseudo-Random Gen / Register
R3 Q R2 R1 Q̄ R0
+1 incrementer logic
C3 O C2 C1 Ō C0
Ne Sequence
Li,Ac,Gc, Wc,Pr,2i Sequence
Controls
O(N-1..0)
CTRL (3): (0): NeU/D (1): 2iU/D, LiU/D, Wc, Gc, Pr (2): 2iU/D, AcQ0,
CTRL (i): (0): NeU/D (1): AcQ0 (2): 2iU/D, LiU/D, Wc, Gc, Pr (3): 2iU/D
CTRL (0): (0): NeU/D (1,2): 2iU/D (3): 2iU/D,AcQ0 (4): 2iU/D, LiU/D, Wc, Gc, Pr
MUX3
CTRL3
O3
MUX(i)
CTRL(i)
O(i)
MUX0
CTRL(0)
O0

Figure 7.18: Li, Ac, Ne, Gc, Wc, Pr and 2i (ALL) AddrGen

'2' and '3'. Alternatively, the generation of the 'Next Sequence' in the $\Uparrow$ or $\Downarrow$ is done via mux inputs '0' and '1'.

7.4.7 Address generator summary

Finally, the AddrGen capable of generating all considered CMs in this section (Li, Ac, Ne, Gc, Wc, Pr and 2i CMs) is also included for comparison, and to illustrate the effectiveness of the new AddrGen implementation method. Figure 7.18 shows its implementation, and it is referred as **ALL** AddrGen. It consists of a LFSR which can be configured both as a Pseudo-random generator and a typical register, with the outputs R_3, R_2, R_1 and R_0; a '+1 incrementer logic' which is the combinational Up-only counter, with the outputs C_3, C_2, C_1 and C_0; and a multiplexer network with the outputs O_3, O_2, O_1 and O_0. The multiplexer consists of $N = 4$ multiplexer, one for each bit. The details of them are shown at the bottom of the Figure 7.18, while their control signals are listed in the upper part of the figure.

Figure 7.19 depicts the area required for each of the CMs covered in this section; for the completeness, the Pseudo-random (Pr) CM (see column 'Pr' in Table 7.1) is also included.

The figure shows that the area required by the 'ALL' AddrGen is 2.42 to 2.95 times the area of the Li AddrGen, depending on the size of N (the larger N, the smaller the size of the ALL AddrGen). On the other hand, the ALL AddrGen requires only 40% of the area required by a brute-force implementation; e.g., for $N = 24$, the ALL AddrGen requires 1054 gates, as compared with 3070 gates for the brute-force implementation of

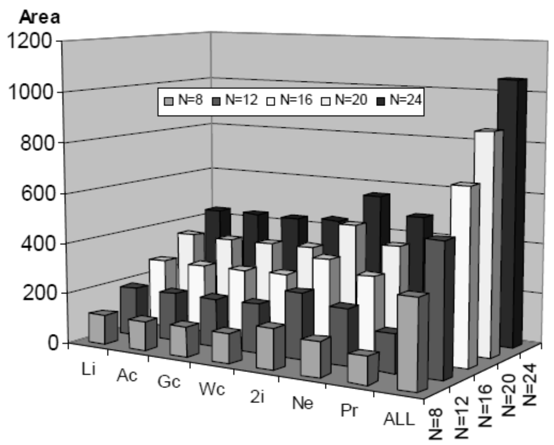


Figure 7.19: Area for different AddrGens

the Address Generator (see Section 7.3.2).

7.5 Experimental Results

This section presents and discusses the GME MBIST Engine experimental results. As the synthesis software tool, Synopsys Design Compiler version D-2010.03 for suse32 was used with its graphical user interface Design Vision. As the technology library, UMC L90 1P9M Standard Performance Low-K Library (FSD0A_A_Generic_Core) was used.

By default, GME MBIST Engine was synthesized for a 16 K x 16-bits memory configuration, with 7-bit row and column addresses, each. Number of Hammer operations and Butterfly Max Distance were chosen as 1000 [98] and 4, respectively. Size of the Command Memory is 64 nibbles of 4-bit, and the clock period was taken as 2 ns (500 MHz) under worst case conditions (WCC). In addition, GME MBIST consists of diagnosis capability. If it is not said reverse, those are the configurations during experiments.

Seven experiments will be presented:

1. **Area vs. flexibility**: investigates that how the GME MBIST area is impacted by the flexibility.

2. **Frequency vs. flexibility**: investigates that how the GME MBIST maximum frequency is impacted by the flexibility.

3. **Area of sub-blocks**: investigates the area percentages of GME MBIST sub-blocks.

4. **Area vs. memory size increase**: investigates that how the GME MBIST area is impacted by the increasing memory size.

5. **At-speed testing**: investigates the at-speed testing capability of GME MBIST.

6. **Area vs. Command Memory size**: investigates how the GME MBIST area is impacted by the Command Memory size.

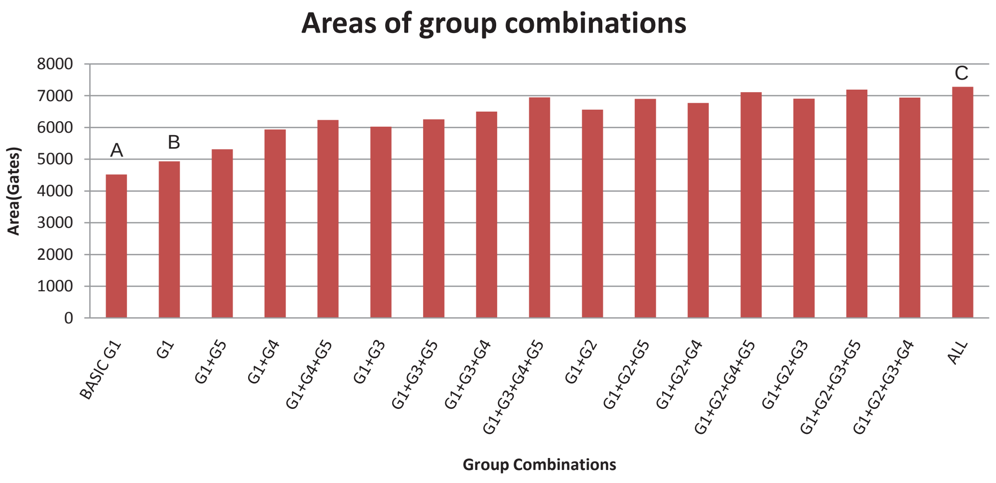


Figure 7.20: GME MBIST area vs. flexibility

7. **Area vs. Diagnosis capability**: investigates how the GME MBIST area is impacted by the diagnosis capability.

7.5.1 Area vs. flexibility

To investigate the impact of the flexibility on the GME MBIST area overhead, five memory test groups are considered as:

- *Group 1.* March test algorithms : This group only consists of the linear March tests (e.g., March C+, March SS, BLIF).
- *Group 2.* Galloping/Walking test algorithms: This group only consists of Galloping and Walking tests (i.e., GalPat, GalRow, GalCol, Gal9R, Walking 1/0, WalkRow, WalkCol).
- *Group 3.* Butterfly test algorithm: This group only contains the Butterfly test.
- *Group 4.* Hammer test algorithms: This group only consists of the Hammer tests (e.g., HamRh, HamWDhrc).
- *Group 5.* Delay test algorithms: This group only consist of the GMEs with a delay time of 50 ms and 100 ms.

By default, each group consists of the $\Updownarrow(wD)$ and $\Updownarrow(rD)$ GMEs.

Group combination names refer to the group names that they include. For example, *G1* represents the implementation of only Group 1, whereas *G1+G2* to combination of the Group 1 and Group 2. For example, *G1* to the implementation for only the Group 1 March algorithms; *G1+G2* for the combination of March and Galloping/Walking algorithms, together. *ALL* is the case where GME MBIST Engine has the full capability.

Table 7.4 shows the area results vs. flexibility. The first column lists the group combinations; the second column gives the area results in terms of μm^2; the third column

Table 7.4: GME MBIST area vs. flexibility

Group Combinations	Area (μm^2)	Area (gates)
BASIC G1	18091	4523
EXTENSION		
G1	19747	4937
G1+G5	21250	5312
G1+G4	23742	5936
G1+G4+G5	24938	6234
G1+G3	24092	6023
G1+G3+G5	25020	6255
G1+G3+G4	25996	6499
G1+G3+G4+G5	27799	6950
G1+G2	26252	6563
G1+G2+G5	27595	6899
G1+G2+G4	27097	6774
G1+G2+G4+G5	28434	7108
G1+G2+G3	27630	6908
G1+G2+G3+G5	28764	7191
G1+G2+G3+G4	27775	6944
ALL	29121	7280

has the normalized area results in terms of 2-input NAND gate. *BASIC G1* corresponds to the G1 implementation with the basic command set. Rest of the combinations were built with the extension command set.

From Figure 7.20, we can conclude the followings:

- The BASIC G1 implementation, BAR A, has an area of 4.5 K gates.
- The EXTENSION G1, BAR B, has an area of 4.9 K gates and requires 9% more area than the BASIC G1.
- The EXTENSION ALL, BAR C, has an area of 7.2 K gates and requires 47.5% more area than the BASIC G1.

To sum up, the correlation between the area overhead and the flexibility was clearly shown by this experiment. Increasing flexibility results in a higher area overhead.

7.5.2 Frequency vs. flexibility

To investigate the impact of the flexibility on the GME MBIST maximum frequency, same memory test groups are used as in the previous experiment.

This experiment concludes the following:

- The EXTENSION G1, BAR B, has a maximum frequency of 714 MHz.
- The EXTENSION ALL, BAR C, has a maximum frequency of 500 MHz which is about 30% slower than the EXTENSION G1.

To sum up, the correlation between the area overhead and the maximum frequency was clearly shown by this experiment. Increasing flexibility results in a lower maximum frequency.

Table 7.5: Area of each sub-block

Module	Area (μm^2)	Area (gates)
GME MBIST Engine	29121	7280
GME MBIST Processor	11170	2793
Command Memory	8332	2083
Controller	2490	623
Registers	348	87
GME MBIST Wrapper	17951	4488
Registers	392	98
Address Generator	13506	3377
Data & Control Generator	1429	357
ORA	1773	443
Multiplexing Unit	851	213

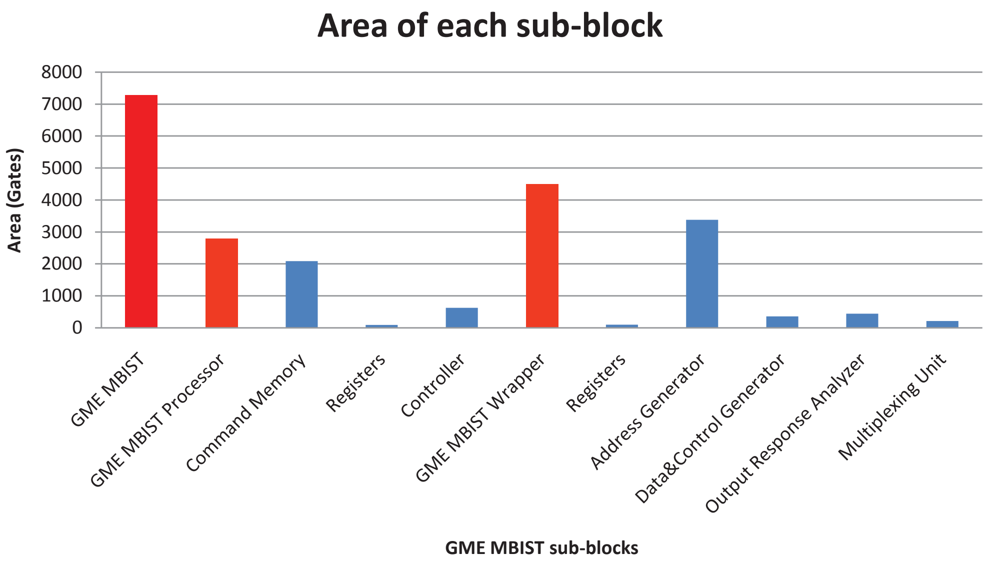


Figure 7.21: Area of each sub-block

7.5.3 Area of each sub-block

To investigate the area overhead of each sub-block in GME MBIST, system was synthesized at 500 MHz for EXTENSION ALL case.

Table 7.5 lists the area for each sub-block. Figure 7.21 visualizes the normalized area overhead. GME MBIST is shown as the red bar. It has two main sub-blocks: GME MBIST Processor and GME MBIST Wrapper. They are shown as the orange bars.

Figure 7.22 concludes the followings:

- GME MBIST is shared between the GME MBIST Processor (38%) and GME MBIST Wrapper (62%).
- GME MBIST Wrapper roughly has an area of 1.6 times of GME MBIST Processor.
- GME MBIST is mainly dominated by the Address Generator (46%) and the Command Memory (29%).
- GME MBIST Processor is mainly dominated by the Command Memory (75%).

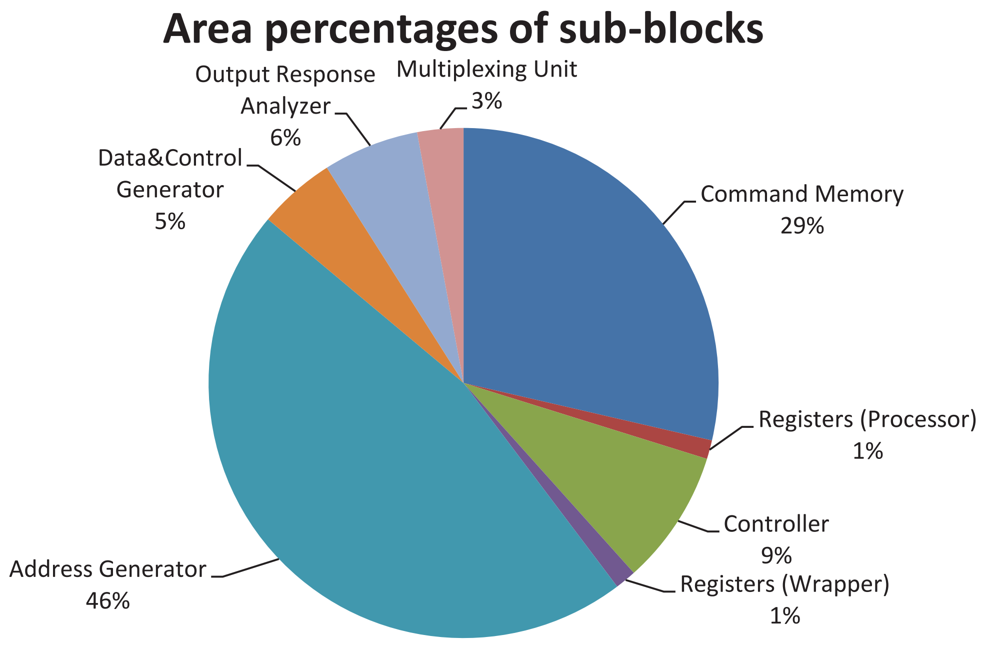


Figure 7.22: Area percentages of each sub-block

Table 7.6: GME MBIST area and area overhead for different memory size configurations

Memory Configuration	GME MBIST area		Memory cell array area	Area OH
(x 16-bit)	(μm^2)	(gates)	(gates)	(%)
16K	28166	7042	76022	9.26
64K	29688	7422	304087	2.44
256K	31283	7821	1216348	0.64
1M	32817	8204	4865393	0.17
4M	34219	8555	19461571	0.04
16M	35943	8986	77846282	0.01

- GME MBIST Wrapper is mainly dominated by the Address Generator (75%).

To sum up, the GME MBIST hardware is dominated by two sub-blocks: the Address Generator (46%) and the Command Memory (29%). Area overhead percentage of each sub-block was clearly shown by this experiment. To further optimize the area overhead, future work should focus on those two sub-blocks.

7.5.4 Area vs. memory size increase

To investigate the impact of memory size increase on the MBIST area overhead, GME MBIST was synthesized for the a number of memory configurations.

In Table 7.6, the first column lists the memory size configurations from 16K x 16-bit to 16M x 16-bit. The second column gives the GME MBIST area in terms of μm^2. The third column is the normalized area in terms of 2-input NAND gate. 2-input NAND gate was chosen since the previous studies used it as an area normalization standard. For a fair area comparison, FSD0A_A_SH library was chosen for the memory cell array, since GME

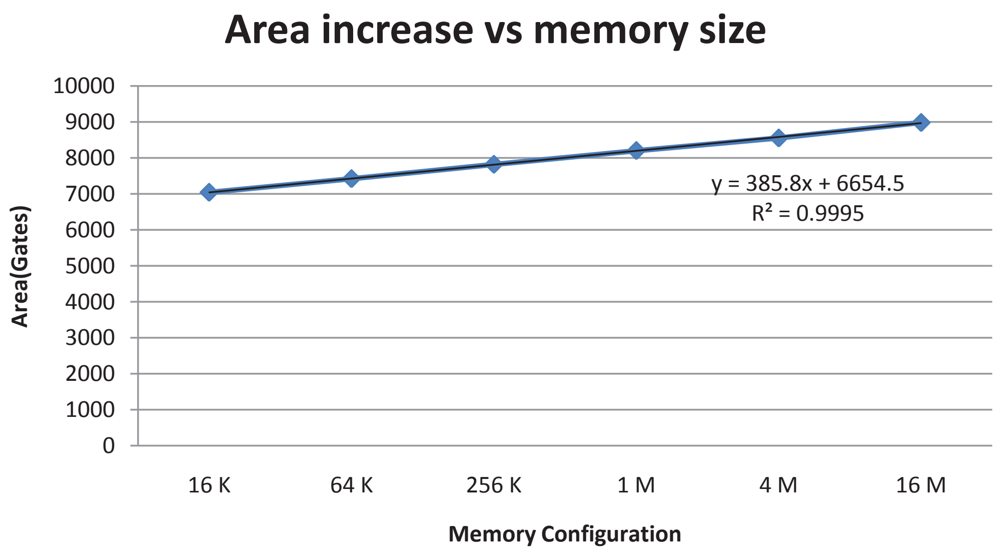


Figure 7.23: Linear area increase of GME MBIST for the increasing memory size

MBIST was synthesized with the logic core of same library (FSD0A_A_Generic_Core). FSD0A_A_SH library is the Faraday 90 nm Synchronous High Density Single-port SRAM Compiler using UMC 90 nm 6TSRAM1.16SPHVT unit cells. This unit cells have an area of 1.16 μm^2 with an operating speed of 425 MHz (Max.) under the worse case conditions for 4K x 16-bit memory configuration [22]. Fourth column shows **ONLY** the normalized area of the memory cell array (i.e., excluding Row/Column Decoders, Read/Write Circuitries, etc.). The last column gives the area overhead in terms of percentages.

For a fair comparison, all of the memory cell arrays and GME MBIST have to be synthesized at the same clock frequency. Since the 256K x 16-bit memory size does not meet a frequency of 500 MHz, it was chosen as 455 MHz for this experiment.

From the figures, the followings are concluded:

- Figure 7.23 shows the linear increase of GME MBIST area for the increasing memory size.
- While the memory size is 4 times larger at each step, GME MBIST area increases roughly 5% (386 gates).
- Figure 7.24 shows the GME MBIST normalized area and the area overhead in percentages.
- While the memory size increases from 16K x 16-bit to 16M x 16-bit, the area overhead falls from 9.26% to 0.01%.
- The memory size goes 4 times high at each step, in parallel GME MBIST area overhead falls 4 times.
- For the high memory sizes, GME MBIST area overhead becomes negligible.

To sum up, the area increase of the GME MBIST is highly linear for exponentially increasing memory sizes. The relation between the GME MBIST area/area overhead

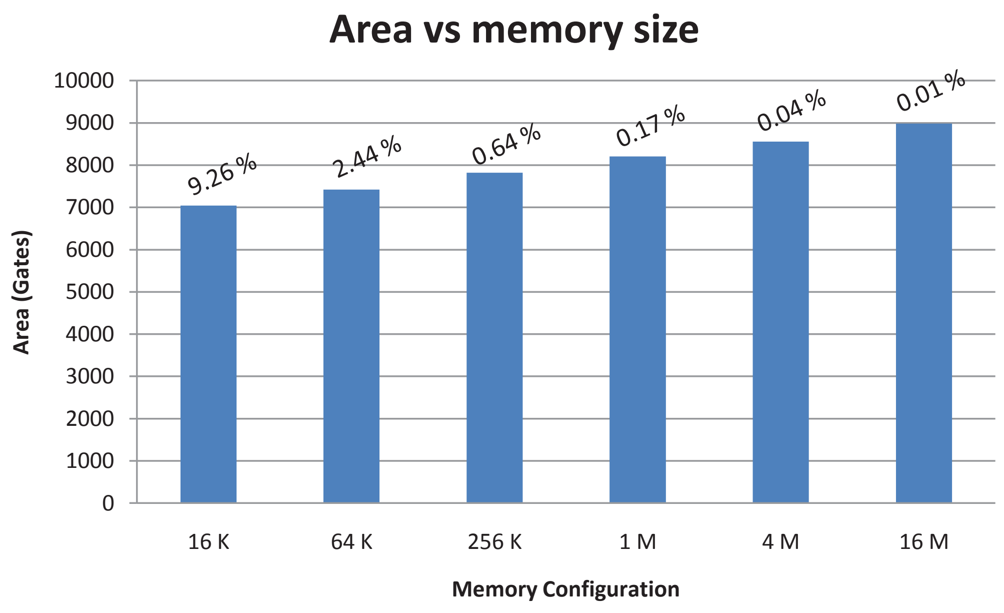


Figure 7.24: GME MBIST area and area overhead for the increasing memory size

Table 7.7: GME MBIST area with/without diagnosis capability for different Command Memory and memory size

	GME MBIST area (gates)	
Command Memory configuration	without Diagnosis	with Diagnosis
16K x 16-bit memory		
Command Memory 8x4-b	4972	5359
Command Memory 16x4-b	5225	5616
Command Memory 64x4-b	6813	7202
8K x 32-bit memory		
Command Memory 8x4-b	5097	5682
Command Memory 16x4-b	5361	5943
Command Memory 64x4-b	6956	7535

and memory size was shown by this experiment.

7.5.5 At-speed testing

To investigate the at-speed testing capability of GME MBIST, it was synthesized and resulting clock frequency was compared with [22].

The clock frequency of a 4K x 16-bit SRAM was reported as 425 MHz for the FSD0A_A_SH library [22]. For a fair comparison, GME MBIST EXTENSION ALL case was synthesized under the same configuration. Resulting clock frequency of the GME MBIST was 555 MHz which is higher than the Faraday's SRAM.

To conclude, this experiment showed that GME MBIST has the at-speed testing capability.

7.5.6 Area vs. Command Memory size

This experiment investigates the impact of Command Memory size on the GME MBIST area.

Since Command Memory has a significant portion (29 %) of the whole hardware, GME MBIST was synthesized for 8, 16 and 64 nibbles of 4-bits Command Memory sizes. In addition, for a fair comparison with the previous studies, this experiment was done for two different Memory-under-Tests: 16K (7-bit row, 7-bit column) x 16-bit and 8K (9-bit row, 4-bit column) x 32-bit.

Table 7.7 concludes the followings:

- Command Memory of 8 nibbles instead of 64 nibbles results in a GME MBIST area saving of 25%.
- The cost of rising Command Memory size from 16 nibbles to 64 nibbles (1590 gates) is roughly 6 times of the rising from 8 nibbles to 16 nibbles (260 gates).
- Command Memory size is linearly proportional to the area overhead.

To sum up, this experiment showed that the Command Memory size has a significant impact on the GME MBIST area. In addition, without changing the MBIST architecture, a certain Command Memory size may be chosen for the fine tuning of the MBIST area overhead to meet the targeted design specifications.

7.5.7 Area vs. Diagnosis capability

This experiment investigates the impact of the diagnosis capability on the GME MBIST area. When the diagnosis feature is added to GME MBIST, the address and data of the faulty cell are reported out to the user.

Due to the same reason in the previous experiment *Area vs. Command Memory size*, this experiment was done for two different Memory-under-Tests: 16K (7-bit row, 7-bit column) x 16-bit and 8K (9-bit row, 4-bit column) x 32-bit.

Table 7.7 concludes the followings:

- For 16K x 16-bit memory, the diagnosis capability adds around 388 gates to the GME MBIST area. 30-bit information (i.e., 14-bit address and 16-bit data) is sent out for diagnosis.
- For 8K x 32-bit memory, the diagnosis capability costs around 580 gates which is 1.5 times of 16K x 16-bit memory. Reason behind that is in this case 45-bit information (i.e., 13-bit address and 32-bit data) is sent out.

To sum up, the data and address widths linearly affect the GME MBIST area overhead.

7.6 Comparison with the previous studies

This section provides a comparison between the GME MBIST and the previous studies from literature.

Table 7.8: Comparison among the previous programmable MBIST studies

	PMBIST [9] (Appello 03)	FP-MBIST [28] (Du 05)	NPMBIST [82] (Park 09)	General [30] FP-MBIST [29] (Du 06)	GME MBIST (this work)
March	✓	✓	✓	✓	✓
Galloping/	x	✓	✓	✓	✓
Walking	x	✓	✓	✓	✓
Butterfly	x	✓	✓	✓	✓
Hammer	x	x	x	x	✓
Sliding Diag.	x	✓	✓	✓	x
AD Open	x	✓	x	✓	✓
AD Delay	x	x	x	x	✓
Byte WR Enb.	x	x	x	✓	x
Moving Inv.	x	x	x	x	✓
Retention	✓	x	x	x	✓
Multi-level nested loop	x	✓(2)	✓(2)	✓(4)	✓(2)
Data Background	s,cb DB Reg	NA	NA	s,cb,r/c Addr. Unq.	s,cb,r/c
Counting Method	U/D fixed	NA	NA	Lr, Lc, Ld	Lr, Lc, Ld AC, 2^i
Command Memory	43x4-b	22x9-b 8x9-b	8x9-b	8x8-b	64x4-b, 16x4-b
Total bits for March C+	160	144	126	216	36
Technology	0.18 μm	0.13 μm	0.13 μm	0.13 μm, 90 nm	90 nm
Freq (MHz)	40	NA	300	NA, 333 [29]	500
AREA 16Kx16-b	7915 gates	9.2 K 7.9 K gates			7.2 K gates 5.6 K gates
AREA 8Kx32-b			6.4 K gates wo Diagnosis	13.6 K gates [30] 4.9 K gates [29]	5.36 K gates 5.94 K gates 3.4 K gates

Table 7.8 lists the GME MBIST and the previous studies on programmable Memory BISTs. Comparison is based on the main features of a MBIST such as the supported test algorithms, data backgrounds, counting methods, MBIST area overhead etc.

Before starting to the comparison, each of the previous studies will be summarized here:

PMBIST (Appello 03)

PMBIST [9] was designed only for March and Retention tests without nested looping. It can generate the solid, checkerboard or any data background defined at Data Background register during the design step. Counting method is fixed during at the design step (i.e., one of the Lr or Lc). Command Memory consists of 43 words of 4-bit instructions. Implementing March C+ test requires 160 bits of instruction memory. Authors proposes that PMBIST has at-speed testing with 40 MHz for 0.18 μm technology. Its prominent feature is owning a P1500-compliant wrapper that enables the execution and diagnosis of the test from an external ATE through IEEE 1149.1 TAP.

FP-MBIST (Du 05)

FP-MBIST [28], developed by Mentor Graphics, is capable of performing a variety of algorithms (i.e., both linear and non-linear) with two level nested looping. FP-MBIST has a modular design that enables the synthesis for the combinations of memory test groups. Supported data backgrounds, counting methods and clock frequency were not mentioned. From the supported algorithms, one can conclude it is capable of fast-row/column and diagonal addressing. Two different Command Memory sizes were reported: 22 words and 8 words of 9-bit instructions. Implementing March C+ test requires 144 bits of instruction memory. It supports full-speed testing by pre-fetching the instructions, meaning that it does not lose any cycles between March operations. It provides a wide memory test menu to the user with diagnosis feature.

NPMBIST (Park 09)

Non-linear PMBIST [82] supports both the linear and non-linear test algorithms. It is able to 2-levels nested looping. Data background and counting method were not mentioned. Command Memory has 8 words of 9-bit instructions. Implementing March C+ algorithm requires 126 bits of instruction memory. Maximum frequency is 300 MHz with TSMC 0.13 μm.

General FP-MBIST (Du 06)

This work [30] is the generalized version of FP-MBIST (Du 05) [28] for an unlimited level of nested looping. For the implementation, they limited themselves up to 4 levels nested looping. General FP-MBIST supports a variety of algorithms. It has solid, checkerboard, row/column stripes, and address unique data backgrounds with fast-row/column, only row/column or diagonal addressing methods. Command Memory has 8 words of 8-bit instructions. Implementing March C+ etst requires is 216 bits of instruction memory. It was implemented at 0.13 μm technology. Frequency was not mentioned at this study, however at [29], same architecture for March-only capability (single level) was synthesized at 90 nm. Resulting frequency was 333 MHz. General FP-MBIST [30] may have the same frequency or most probably it is slower than that. Number of nested loop level, data backgrounds, and addressing scheme within a loop are orthogonal to each other.

GME MBIST (this work)

GME MBIST supports a high variety of test algorithms. For the delay tests, it consists of a programmable delay register enabling up to 100 ms delays at 500 MHz clock frequency. GME MBIST has a modular design that enables the synthesis for the combinations of memory test groups. 2-levels nested looping is supported. It orthogonally generates the address order (Up/Down), data background patterns (solid, checkerboard, row/column stripes), counting methods (fast-row/column/diagonal, address complement, 2^i). By default, the Command Memory size is 64 nibbles of 4-bit. Instructions have variable-lengths to further save from the command storage. Implementing March C+ algorithm requires only to 36 bits. Implementation at 90 nm technology resulted in a frequency of 500 MHz. It has at-speed testing and diagnosis.
Here 7 comparisons between GME MBIST and the previous studies will be presented as:

1. **Memory test algorithm support**: compares in terms of the supported memory test algorithms.

2. **Multi-level nested looping support**: compares in terms of the number of nested loop levels.

3. **Data backgrounds and counting methods**: compares in terms of the data backgrounds and counting methods.

4. **Instruction size and coding efficiency**: compares the in terms of the instruction size and coding efficiency.

5. **Diagnosis capability**: compares in terms of the diagnosis capability.

6. **Frequency**: compares in terms of the achieved frequency.

7. **Area overhead**: compares in terms of the MBIST area overhead.

7.6.1 Memory test algorithm support

PMBIST supports March-only algorithms and retention tests. It manages the delay tests by no operation instructions. Therefore, for a long delay time (e.g., a few hundred milliseconds), instruction memory gets too high. Whereas GME MBIST has a programmable delay register supporting any delay amount. FP-MBIST, NPMBIST and General FP-MBIST do not support Hammer, AD delay, MOVI or retention tests.

To conclude, GME MBIST is superior than the previous studies in terms of supported algorithms, since it provides critical memory tests.

7.6.2 Multi-level nested looping support

PMBIST does not have multi-level nested looping capability. FP-MBIST, NPMBIST and General FP-MBIST have 2, 2 and 4-levels nested loops, respectively. GME MBIST supports 2-levels nested loops. In fact, more than 2-levels is open to discussion; whether it is necessary or it is over design.

To conclude, GME MBIST is equal compared to the previous studies in terms of multi-level nested looping capability.

7.6.3 Data backgrounds and counting methods

For the data backgrounds, PMBIST has solid, checkerboard and a programmable data background feature. FP-MBIST and NPMBIST do not mention the supported data backgrounds. General FP-MBIST has an address unique data background in addition to solid, checkerboard, row/column stripes. It enables to specify a unique data background only for a specific address location. GME MBIST only does not provide address unique data background.

For the counting methods, PMBIST has a counter that is fixed during the design step. FP-MBIST and NPMBIST do not mention the supported addressing schemes.

General FP-MBIST has fast-row/column and diagonal schemes. Except GME MBIST, the previous studies do not provide address complement (AC) or 2^i counting methods.

To conclude, GME MBIST is superior compared to the previous implementations in terms of the supported counting methods. In addition, GME MBIST offers critical data background patterns except the address unique pattern.

7.6.4 Instruction size and coding efficiency

PMBIST, FP-MBIST, NPMBIST and General FP-MBIST have instruction sizes of 4, 9, 9 and 8-bit; they require a total of 160, 144, 126 and 216 bits to implement March C+ test, respectively. Whereas, GME MBIST has variable-length commands. Meaning that the opcode length is inversely related to the usage frequency of the command. Moreover, GME MBIST only requires 36 bits to implement March C+ test. This is due to that our proposed MBIST hardware is based on the novel concept of GME, whereas previous studies (i.e., FP-MBIST, NPMBIST and General FP-MBIST) are operation-based MBISTs. Therefore, they have to define the each memory operation; resulting in high amount of test code. On the other hand, GME concept compacts the information of several memory operations into a single GME and saves from the test code.

To conclude, GME MBIST has better coding efficiency and requires less Command Memory space due to the variable-length commands and the novel concept of GME.

7.6.5 Diagnosis capability

PMBIST has a P1500 compliant wrapper which can communicate with a IEEE 1149.1 TAP controller for diagnosis purposes. NPMBIST does not support diagnosis, whereas the rest and our GME MBIST have diagnosis feature.

This comparison shows that GME MBIST is equal to the previous studies in terms of the diagnosis capability.

7.6.6 Frequency

Within the previous studies, only [29] reported a frequency of 333 MHz at 90 nm for a March-only General FP-MBIST. On the other hand, GME MBIST EXTENSION ALL case has a frequency of 500 MHz for the same technology node. Furthermore, when General FP-MBIST is synthesized for more algorithms, its resulting frequency will be most probably lower than the reported value. Therefore, GME MBIST is superior to the previous studies in terms of the frequency.

7.6.7 Area overhead

For a fair comparison with the previous studies, GME MBIST was synthesized for two memory sizes: 16K x 16-b and 8K x 32-b memories. In addition, some of the previous studies reported MBIST area for a Command Memory size of 43x4-b, 22x9-b, 8x9-b, 8x8-b with or without diagnosis.

PMBIST reported an area of 7951 gates for a 16K x 16-bit memory size with a CM size of 43x4-b (162 bits). GME MBIST has an area of 7202 gates for the same memory size with a CM size of 64x4-b (256 bits). Although our Command Memory is larger and GME MBIST is at its full capability, we have a lower area of 749 gates.

FP-MBIST reported an area of 9.2 K gates with a CM size of 22x9-b (198 bits); and 7.9 K with a CM size of 8x9-b (72 bits) for a 16K x 16-bit memory size. For the comparison, GME MBIST has an area of 7.2 K gates with a CM size of 64x4-b (256 bits); and an area of 5.6 K gates with a CM size of 16x4-b (64 bits) for the same memory size. The area of GME MBIST is 2.7 K and 2.3 K gates lower than the FP-MBIST.

NPMBIST reported an area of 6.4 K gates for a 8K x 32-bit memory size without diagnosis. The size of CM was not mentioned, however they apply the dynamic loading as General FP-MBIST where CM sizes is 8x8-b (64 bits). Thus, GME MBIST configuration was selected for a 8K x 32-bit memory size with a CM size of 16x4-b (64 bits). GME MBIST has an area of 5.36 K gates which is 1.04 K gates lower than the area of NPMBIST.

General FP-MBIST reported an area of 13.6 K gates for a 8K x 32-bit memory size with a CM size of 8x8-b (64 bits). GME MBIST has an area of 5.94 K gates for the same memory size with a CM size of 16x4-b (64 bits). The area of GME MBIST is 7.66 K gates lower than the General FP-MBIST.

To conclude, this comparison shows that GME MBIST is superior to the previous studies in terms of area under several configurations (e.g., different memory size, Command Memory size, diagnosis capability).

7.7 Conclusion

This section concludes the points that GME MBIST is superior to the previous studies.

Experiments results and the comparison with the previous studies shows that GME MBIST is superior on the followings:

- GME MBIST is more flexible. Meaning that it supports more memory test algorithms.
- GME MBIST is equal in terms of the supported number of nested loop levels.
- GME MBIST is far better in terms of the generated counting methods.
- GME MBIST is equal in terms of the generated data background patterns. Only the address unique data background pattern is not supported.
- GME MBIST is far better in terms of the instruction coding and Command Memory usage. This is mainly due to the variable-length commands and the novel GME concept.
- GME MBIST is equal in terms of the diagnosis capability.
- GME MBIST is better in terms of the area overhead.
- GME MBIST is unique in terms of providing the orthogonal algorithm stresses.

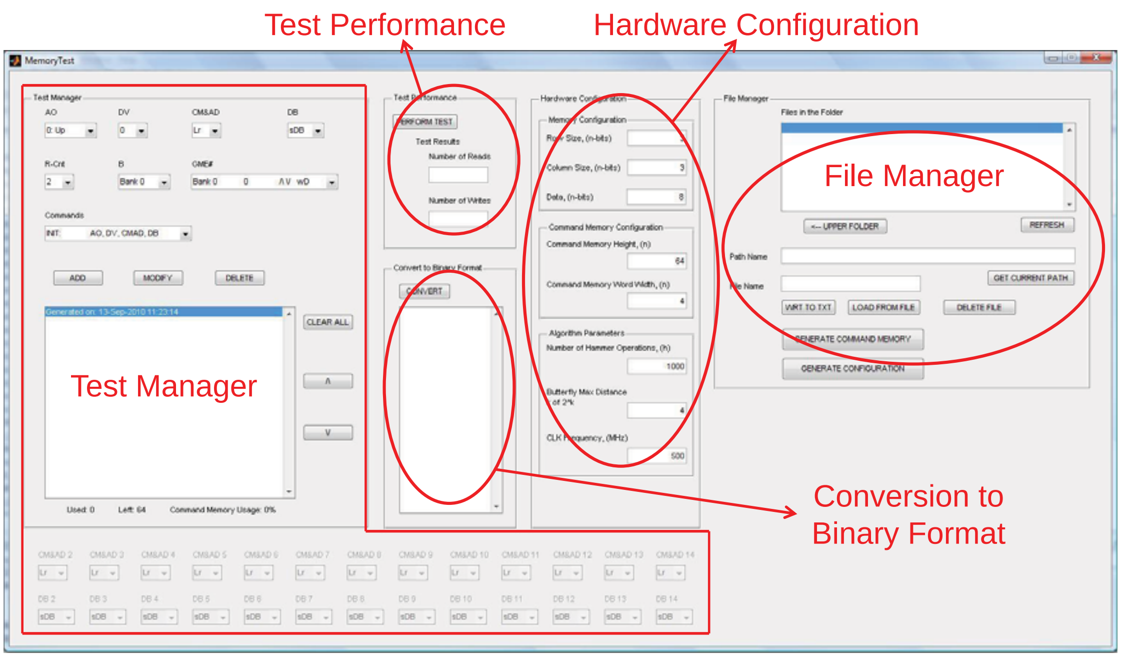
Therefore, we can conclude that the design goals: 1) high flexibility, 2) at-speed testing, 3) low area overhead, 4) linear area increase with the increasing memory size, 5) user-programmability, 6) full-speed testing, 7) orthogonal algorithm stresses, 8) diagnosis capability have been met.

GME MBIST Test Manager 8

This chapter introduces a graphical user interface (GUI) that simplifies the implementation of memory tests. Main aim is to guide the user for operating the GME MBIST. This chapter is organized as follows. Section 8.1 presents the test manager where the memory tests are programmed. Section 8.2 explains the part where the performance of a test is checked in high-level. Section 8.3 and 8.4 continue with the generation of test microcode and the configuration parameters for the hardware synthesis, respectively. Section 8.5 concludes with the file manager system that saves/loads the memory test files.

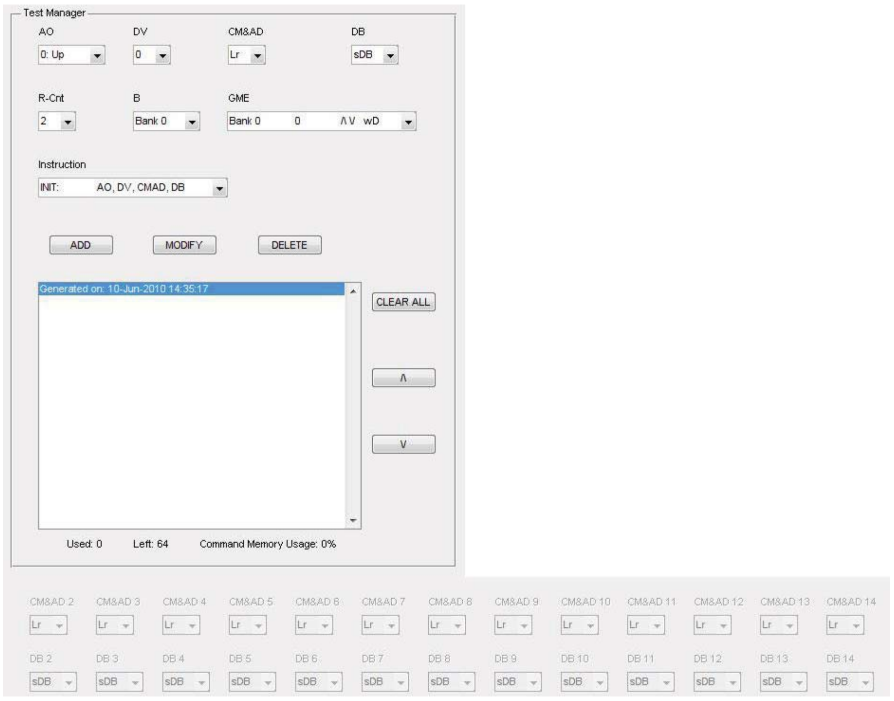
Figure 8.1 shows that the GUI has five sub-blocks as:

1. Test Manager: generates the memory tests by using the GME MBIST commands.
2. Test Performance: checks the performance of the defined memory test in Test Manager.
3. Convert to Binary Format: generates the microcode file for the Command Memory. It converts the defined memory test in Test Manager to the binary format.



AO: 0: Up; DV: 0; CM&AD: Lr; DB: sDB
R-Cnt: 2; B: Bank 0; GME#: Bank 0 0 ΛV wD
Commands: INIT: AO, DV, CMAD, DB
Generated on: 13-Sep-2010 11:23:14
Used: 0 Left: 64 Command Memory Usage: 0%
Number of Reads
Number of Writes
Row Size, (n-bits)
Column Size, (n-bits) 3
Data, (n-bits) 8
Command Memory Height, (n) 64
Command Memory Word Width, (n) 4
Number of Hammer Operations, (h) 1000
Butterfly Max Distance of 2^k 4
CLK Frequency, (MHz) 500
CM&AD 2 Lr, CM&AD 3 Lr, CM&AD 4 Lr, CM&AD 5 Lr, CM&AD 6 Lr, CM&AD 7 Lr, CM&AD 8 Lr, CM&AD 9 Lr, CM&AD 10 Lr, CM&AD 11 Lr, CM&AD 12 Lr, CM&AD 13 Lr, CM&AD 14 Lr
DB 2 sDB, DB 3 sDB, DB 4 sDB, DB 5 sDB, DB 6 sDB, DB 7 sDB, DB 8 sDB, DB 9 sDB, DB 10 sDB, DB 11 sDB, DB 12 sDB, DB 13 sDB, DB 14 sDB

Figure 8.1: Graphical User Interface for GME MBIST; Test Manager, Test Performance, Conversion to Binary Format, Hardware Configuration, and File Manager


Test Manager

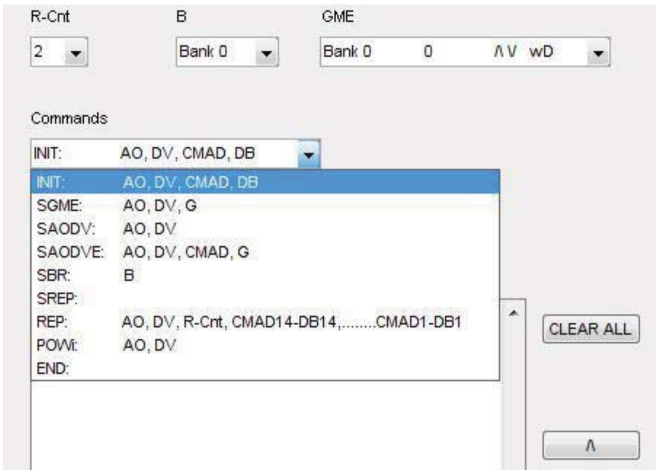
AO: 0: Up
DV: 0
CM&AD: Lr
DB: sDB
R-Cnt: 2
B: Bank 0
GME: Bank 0 0 /\ V wD
Instruction: INIT: AO, DV, CMAD, DB

Generated on: 10-Jun-2010 14:35:17

Used: 0 Left: 64 Command Memory Usage: 0%

CM&AD 2 … CM&AD 14: Lr
DB 2 … DB 14: sDB

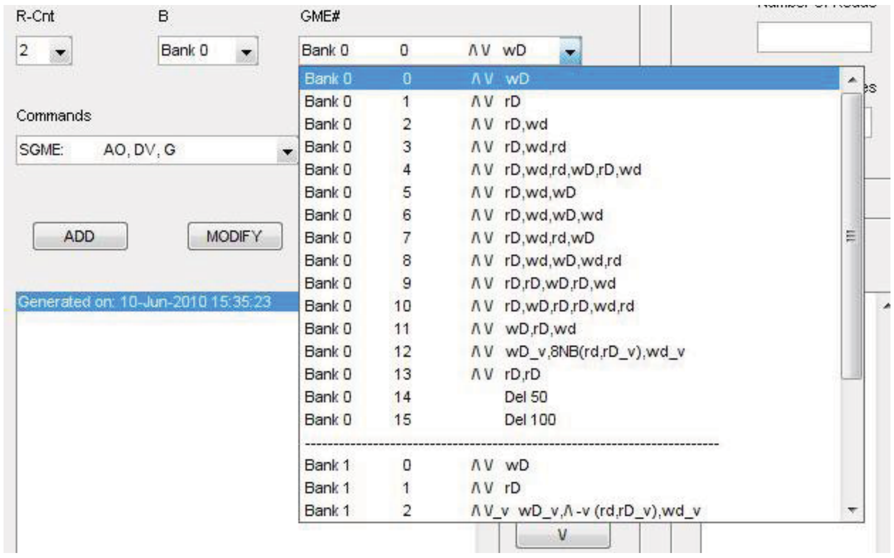
Figure 8.2: Test Manager


R-Cnt: 2
B: Bank 0
GME: Bank 0 0 /\ V wD
Commands: INIT: AO, DV, CMAD, DB

```
INIT:     AO, DV, CMAD, DB
SGME:     AO, DV, G
SAODV:    AO, DV
SAODVE:   AO, DV, CMAD, G
SBR:      B
SREP:
REP:      AO, DV, R-Cnt, CMAD14-DB14,........CMAD1-DB1
POWi:     AO, DV
END:
```

Figure 8.3: Commands pop-up menu showing all commands with their required parameters

4. Hardware Configuration: generates the file that contains the synthesis parameters.
5. File Manager: manages the memory test file loading/saving.



```
R-Cnt: 2
B: Bank 0
GME#: Bank 0    0    ΛV  wD
Commands: SGME:    AO, DV, G
Generated on: 10-Jun-2010 15:35:23

Bank 0    0    ΛV  wD
Bank 0    1    ΛV  rD
Bank 0    2    ΛV  rD,wd
Bank 0    3    ΛV  rD,wd,rd
Bank 0    4    ΛV  rD,wd,rd,wD,rD,wd
Bank 0    5    ΛV  rD,wd,wD
Bank 0    6    ΛV  rD,wd,wD,wd
Bank 0    7    ΛV  rD,wd,rd,wD
Bank 0    8    ΛV  rD,wd,wD,wd,rd
Bank 0    9    ΛV  rD,rD,wD,rD,wd
Bank 0    10   ΛV  rD,wD,rD,rD,wd,rd
Bank 0    11   ΛV  wD,rD,wd
Bank 0    12   ΛV  wD_v,8NB(rd,rD_v),wd_v
Bank 0    13   ΛV  rD,rD
Bank 0    14       Del 50
Bank 0    15       Del 100
--------------------------------------------------------------
Bank 1    0    ΛV  wD
Bank 1    1    ΛV  rD
Bank 1    2    ΛV_v  wD_v,Λ-v (rd,rD_v),wd_v
```

Figure 8.4: GME# pop-up menu showing all GMEs listed in Table 6.2

8.1 Test Manager

Test Manager helps user to build own self-defined memory tests. It consists of two main objects: pop-up menus and push buttons.

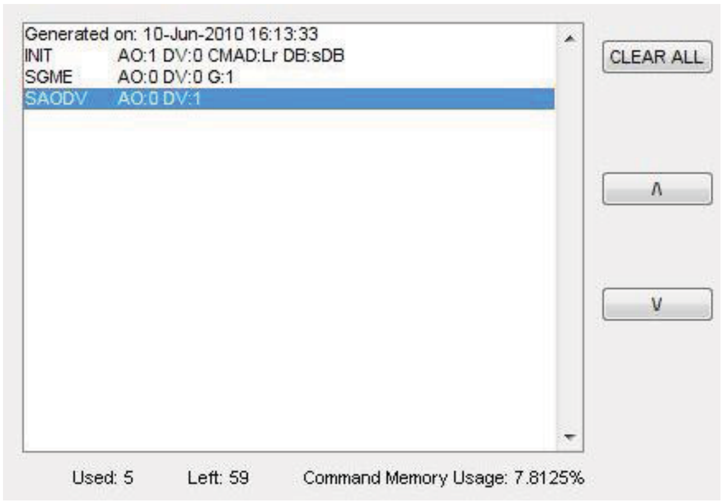
A pop-up menu shows the different options for a certain parameter. A certain parameter can be selected from the list of a pop-up. Figure 8.2 shows that the user may select one of the GME MBIST commands with the required parameters (i.e., AO, DV, CM&AD, DB, B, GME# or RCNT#). For example, when clicked onto the Commands pop-up, all commands and the parameters required by them appear as in Figure 8.3. In addition, Figure 8.4 shows that a certain GME listed in Table 6.2 is specified from the GME# pop-up menu.

Push buttons are used for adding, removing, moving up/down of a selected command. Push buttons are presented in the following as:

- ADD: adds the selected command with its parameters to the specified position on the test listbox.
- MODIFY: modifies (overwrites) the selected command with the new parameters.
- DELETE: deletes the selected command from the test listbox.
- CLEAR ALL: clears all of the test listbox.
- /\: moves up the selected command.
- \/: moves down the selected command.

To further simplify the implementation of a memory test, *Usage Statistics* of the Command Memory are shown at the bottom of the test listbox. Figure 8.5 shows the three main usage information:

- Used: shows the number of occupied lines (4-bit nibbles) in the Command Memory.


Generated on: 10-Jun-2010 16:13:33
INIT AO:1 DV:0 CMAD:Lr DB:sDB
SGME AO:0 DV:0 G:1
SAODV AO:0 DV:1

Used: 5 Left: 59 Command Memory Usage: 7.8125%

Figure 8.5: Usage statistics of the Command Memory

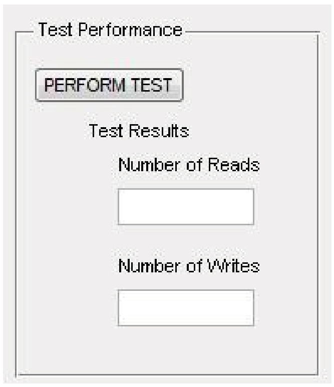


Figure 8.6: Test Performance

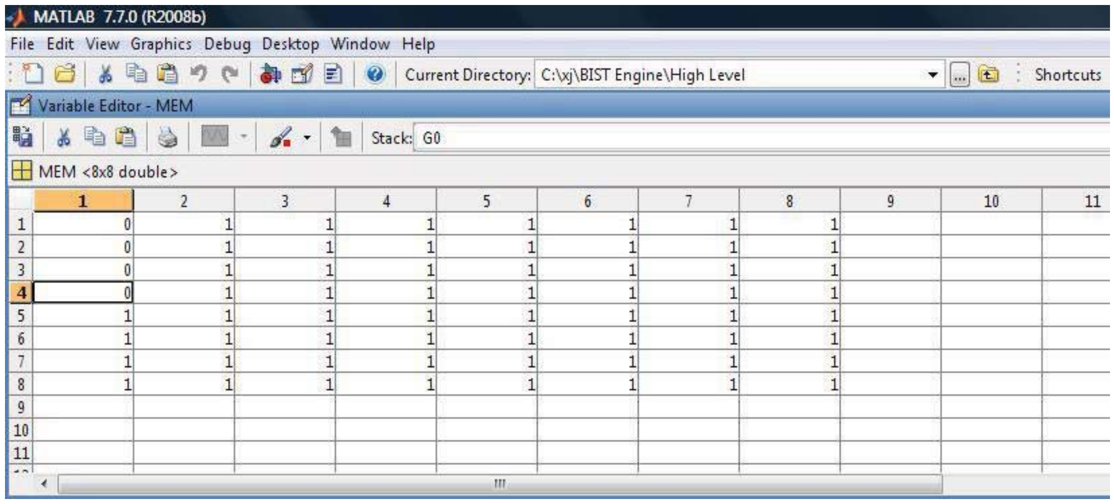
- Left: shows the empty lines in the Command Memory.
- Command Memory Usage: gives the usage percentage. For example, Figure 8.5 shows that *Used* is 5 $(2*2+1*1)$ and *Left* is 59 $(64-5)$. Then, *Command Memory Usage* is simply calculated as 7.8125% $((5/64)*100)$.

8.2 Test Performance

This module executes the defined memory test in high level and reports the number of read/write operations back.

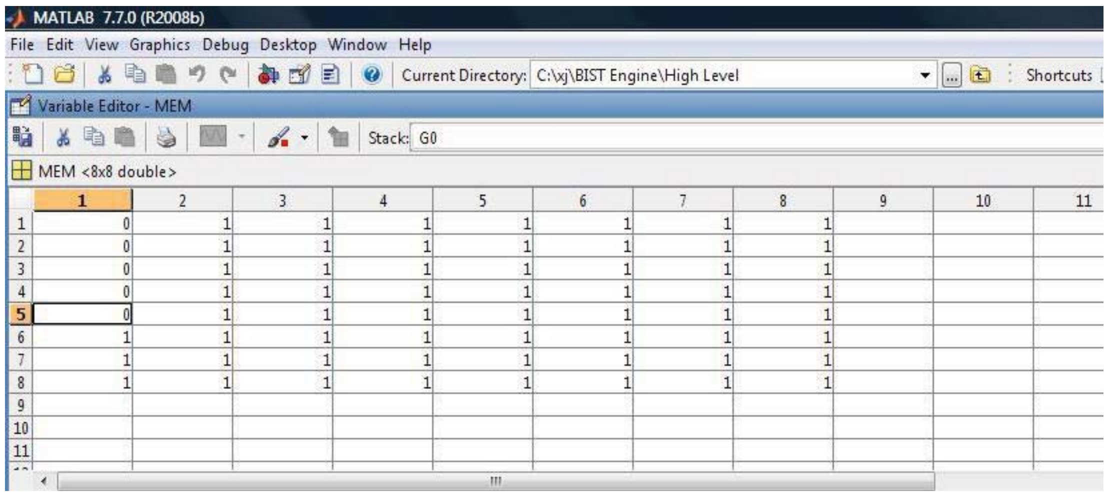
Test Performance reads the test commands specified in the listbox, and performs the test at high-level. Figure 8.6 shows that *Test Results* reports the number of read and write operations.

Furthermore, Test Performance enables to view the state of the memory array at each command step. To do so, the user should open the *MemoryTest.m* file, find the func-


MATLAB 7.7.0 (R2008b)
File Edit View Graphics Debug Desktop Window Help
Current Directory: C:\xj\BIST Engine\High Level
Shortcuts
Variable Editor - MEM
Stack: G0
MEM <8x8 double>

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	1	1	1	1	1			
2	0	1	1	1	1	1	1	1			
3	0	1	1	1	1	1	1	1			
4	0	1	1	1	1	1	1	1			
5	1	1	1	1	1	1	1	1			
6	1	1	1	1	1	1	1	1			
7	1	1	1	1	1	1	1	1			
8	1	1	1	1	1	1	1	1			
9											
10											
11											

Figure 8.7: State of the memory array for Lr, Down addressing with solid DB and DV:0 for an 8 words with 8 columns

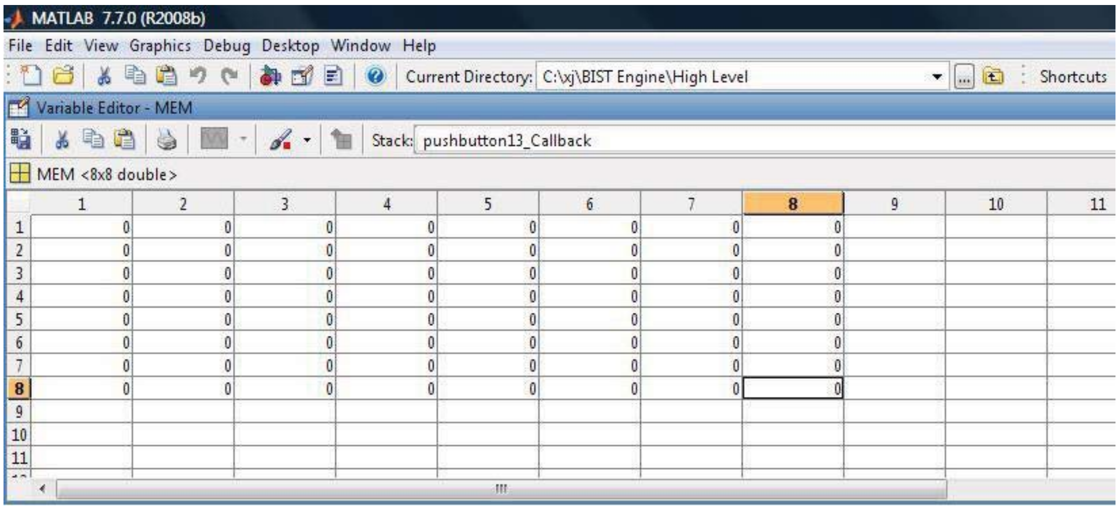

MATLAB 7.7.0 (R2008b)
File Edit View Graphics Debug Desktop Window Help
Current Directory: C:\xj\BIST Engine\High Level
Shortcuts
Variable Editor - MEM
Stack: G0
MEM <8x8 double>

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	1	1	1	1	1			
2	0	1	1	1	1	1	1	1			
3	0	1	1	1	1	1	1	1			
4	0	1	1	1	1	1	1	1			
5	0	1	1	1	1	1	1	1			
6	1	1	1	1	1	1	1	1			
7	1	1	1	1	1	1	1	1			
8	1	1	1	1	1	1	1	1			
9											
10											
11											

Figure 8.8: State of the memory array at next step

tion for the *PERFORM TEST* push button (function pushbutton13_Callback(hObject, eventdata, handles)), and put a breakpoint at the line ”while Line <= size(List,1)”. Then, the variable *MEM* under the MATLAB Workspace window shows the current state of the memory array.

Moreover, to see the internal addressing and read/write data to each of the memory cells, user should get in the function *HL_GME_MBIST()*, and put the breakpoint to the related GME function. For example, Figure 8.7 shows a GME#0: $\Downarrow (w0)$ operation for an 8 x 8 memory array. Lr Down addressing in combination with a solid data background and zero data value are the specified algorithm stresses. Figure 8.8 shows the one further step: victim cell at Row 5 & Column 1 is written 0. Finally, Figure 8.9 shows the memory



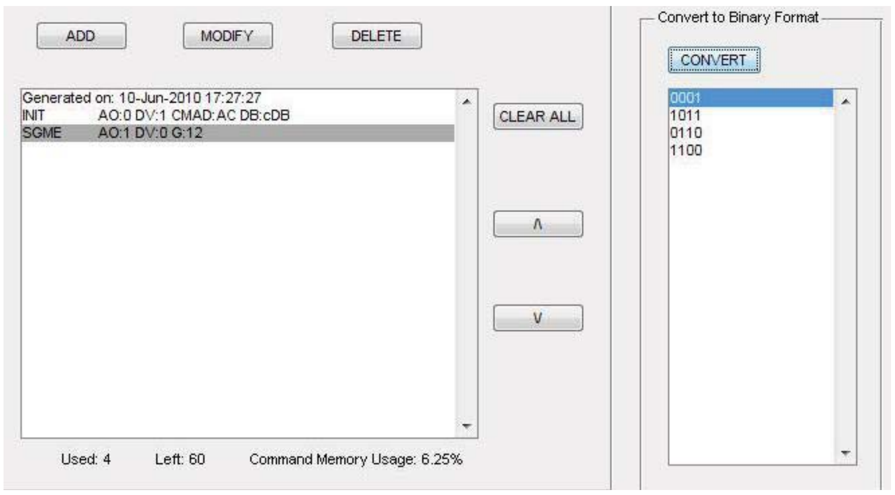
Current Directory: C:\xj\BIST Engine\High Level

Stack: pushbutton13_Callback

MEM <8x8 double>

	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	0	0	0	0			
2	0	0	0	0	0	0	0	0			
3	0	0	0	0	0	0	0	0			
4	0	0	0	0	0	0	0	0			
5	0	0	0	0	0	0	0	0			
6	0	0	0	0	0	0	0	0			
7	0	0	0	0	0	0	0	0			
8	0	0	0	0	0	0	0	0			
9											
10											
11											

Figure 8.9: Final state of the memory array after initialization with DV: 0 and solid DB



```
Generated on: 10-Jun-2010 17:27:27
INIT     AO:0 DV:1 CMAD:AC DB:cDB
SGME     AO:1 DV:0 G:12
```

```
0001
1011
0110
1100
```

Used: 4   Left: 60   Command Memory Usage: 6.25%

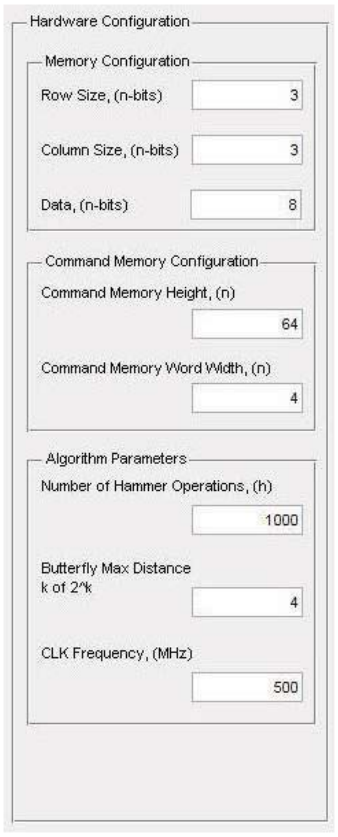
Figure 8.10: Binary converted commands

array initialized with the data value of 0.

An important remark is that, Test Performance models the memory array for the data width of a single bit. For example, an 8 x 8 memory will be an 8x8 2-D array regardless of the number of data bits, as shown in Figure 8.9.

8.3 Convert to Binary Format

This part converts the specified memory test to microcode to be loaded into the Command Memory. Conversion to Binary Format reads the commands from the listbox, and converts them to their binary equivalent in the form of 4-bit nibbles. For example, Figure 8.10 shows that the command ”INIT AO: 0, DV: 1, CMAD: AC, DB: cDB” is converted


Hardware Configuration

Memory Configuration

Row Size, (n-bits) 3

Column Size, (n-bits) 3

Data, (n-bits) 8

Command Memory Configuration

Command Memory Height, (n) 64

Command Memory Word Width, (n) 4

Algorithm Parameters

Number of Hammer Operations, (h) 1000

Butterfly Max Distance k of 2^k 4

CLK Frequency, (MHz) 500

Figure 8.11: Hardware Configuration

as "0001", "1011"; and the command "SGME AO: 1, DV: 0, G: 12" is converted as "0110", "1100". Binary codings of the commands and the algorithm stresses are given at Section 7.2.1.

8.4 Hardware Configuration

This part generates the file that contains the hardware synthesis parameters. Figure 8.11 shows that it consists of the followings:

- Memory Row/Column Size: shows the width of the memory row/column addresses.
- Memory Data Bits Width: shows the memory data width.
- Command Memory Height/Word Width: shows the number of rows and columns of the Command Memory. By default, it has a size of 64 rows x 4-bit.
- Number of Hammer operations (h): from [98], h is 1000 by default.
- Butterfly Max Distance: is the parameter k of 2^k distance for Butterfly test.
- CLK Frequency (MHz): for the GMEs with a delay of 50 and 100 ms, the width of the delay counter is calculated depending on the clock frequency.

All those configuration parameters are read, converted into binary format, and then written into the hardware configuration file by the *GENERATE CONFIGURATION*

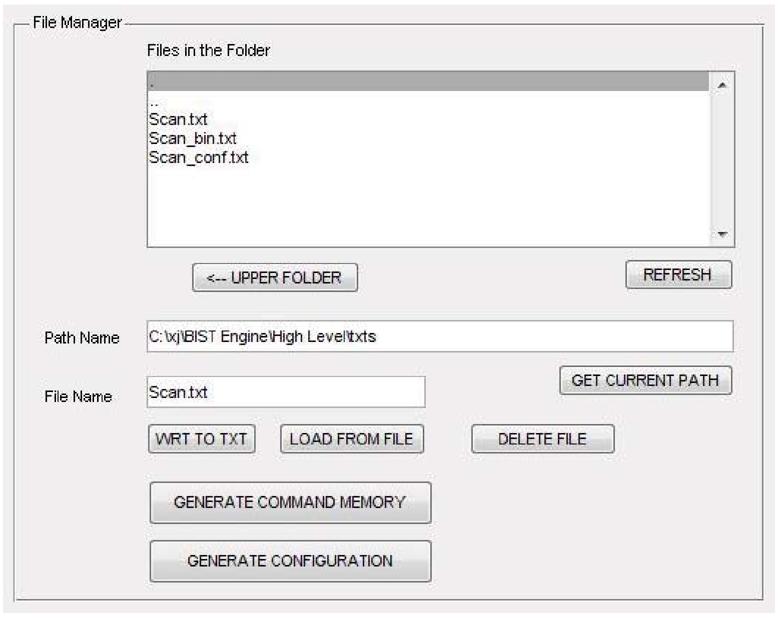


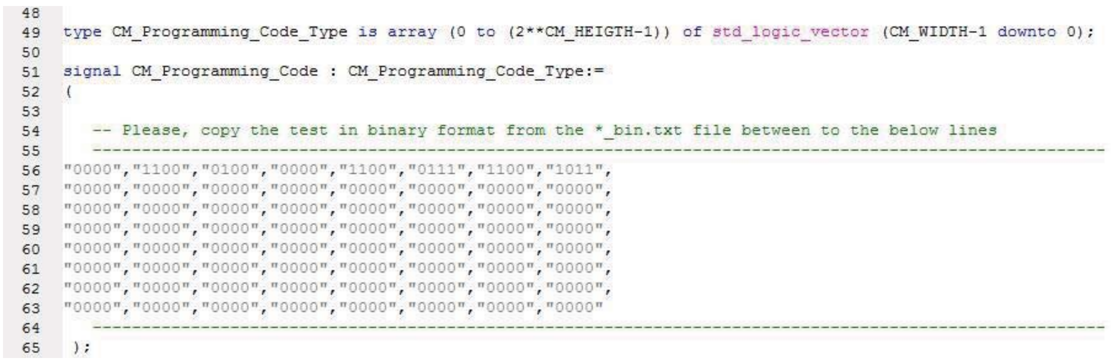
Figure 8.12: File Manager

push button under the File Manager.

8.5 File Manager

File Manager is responsible from the project file management. It can save a specified memory test to the test library; load a memory test from the test library to the Test Manager; or delete a file in the test library. In addition, it generates the files that contain the microcode for the Command Memory, and the parameters for the hardware synthesis. Figure 8.12 shows that it consists of several edit boxes and push buttons as:

- Path Name: shows the path of the current project folder.
- File Name: Name of the file that will be written, loaded from or be deleted is typed into this box. For example, ”Scan.txt”.
- GET CURRENT PATH: automatically writes the pathname of the current project folder into the Path Name box. In further, the content of the current folder is shown under the ”Files in the Folder” listbox.
- <– UPPER FOLDER: goes up to the folder at one higher level and shows the content of the folder.
- WRT TO TXT: reads the test commands from the Test Manager. Without converting into binary, specified test commands are written into the specified file at the ”File Name” box. For example, ”Scan.txt”.
- LOAD FROM FILE: read the file specified at the ”File Name” box, and writes back to the Test Manager. For example, ”Scan.txt”.
- DELETE FILE: deletes the specified file at the ”File Name” box. For example, ”Scan.txt”.

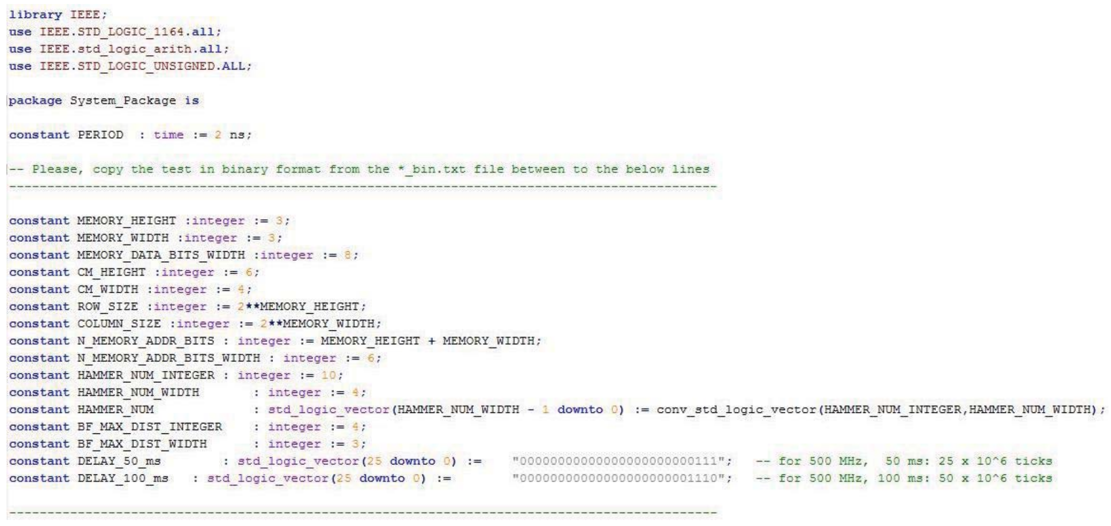


```
type CM_Programming_Code_Type is array (0 to (2**CM_HEIGTH-1)) of std_logic_vector (CM_WIDTH-1 downto 0);

signal CM_Programming_Code : CM_Programming_Code_Type:=
(

   -- Please, copy the test in binary format from the *_bin.txt file between to the below lines
   ------------------------------------------------------------------------------------------------------
"0000","1100","0100","0000","1100","0111","1100","1011",
"0000","0000","0000","0000","0000","0000","0000","0000",
"0000","0000","0000","0000","0000","0000","0000","0000",
"0000","0000","0000","0000","0000","0000","0000","0000",
"0000","0000","0000","0000","0000","0000","0000","0000",
"0000","0000","0000","0000","0000","0000","0000","0000",
"0000","0000","0000","0000","0000","0000","0000","0000",
"0000","0000","0000","0000","0000","0000","0000","0000"
   ------------------------------------------------------------------------------------------------------
 );
```

Figure 8.13: The area to be pasted from "..._bin.txt" to TB_GME_MBIST.vhd



```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package System_Package is

constant PERIOD  : time := 2 ns;

-- Please, copy the test in binary format from the *_bin.txt file between to the below lines
--------------------------------------------------------------------------------------------

constant MEMORY_HEIGHT :integer := 3;
constant MEMORY_WIDTH :integer := 3;
constant MEMORY_DATA_BITS_WIDTH :integer := 8;
constant CM_HEIGHT :integer := 6;
constant CM_WIDTH :integer := 4;
constant ROW_SIZE :integer := 2**MEMORY_HEIGHT;
constant COLUMN_SIZE :integer := 2**MEMORY_WIDTH;
constant N_MEMORY_ADDR_BITS : integer := MEMORY_HEIGHT + MEMORY_WIDTH;
constant N_MEMORY_ADDR_BITS_WIDTH : integer := 6;
constant HAMMER_NUM_INTEGER : integer := 10;
constant HAMMER_NUM_WIDTH       : integer := 4;
constant HAMMER_NUM             : std_logic_vector(HAMMER_NUM_WIDTH - 1 downto 0) := conv_std_logic_vector(HAMMER_NUM_INTEGER,HAMMER_NUM_WIDTH);
constant BF_MAX_DIST_INTEGER    : integer := 4;
constant BF_MAX_DIST_WIDTH      : integer := 3;
constant DELAY_50_ms        : std_logic_vector(25 downto 0) :=    "00000000000000000000000111";   -- for 500 MHz,  50 ms: 25 x 10^6 ticks
constant DELAY_100_ms   : std_logic_vector(25 downto 0) :=        "00000000000000000000001110";   -- for 500 MHz, 100 ms: 50 x 10^6 ticks

--------------------------------------------------------------------------------------------
```

Figure 8.14: The area to be pasted from "..._conf.txt" to System_Package.vhd

- GENERATE COMMAND MEMORY: reads the binary converted test commands from the Convert to Binary Format, and writes them into the file "..._bin.txt". For "Scan.txt", the microcode will be generated in the file "Scan_bin.txt".
- GENERATE CONFIGURATION: reads the parameters under the Hardware Configuration, and writes them into the file "..._conf.txt". For "Scan.txt", synthesis parameters will be generated in the file "Scan_conf.txt".

After generating the microcode and synthesis parameters files, the user should copy and paste the contents of those two files ("..._bin.txt" and "..._conf.txt") to the "TB_GME_MBIST.vhd" and "System_Package.vhd" files under the GME_MBIST_RTL folder, respectively. Figure 8.13 and 8.14 show the regions inside those two VHDL files, to where the copied codes should be pasted.

Conclusion 9

This chapter concludes this thesis study. Section 9.1 mentions the main contributions of this thesis. Section 9.2 suggests the points that may be further studied/improved for future work.

9.1 Contributions

This thesis contributes to the field of memory testing. Specifically, a novel Memory BIST implementation was introduced. The proposed MBIST is based on the concept of Generic and Orthogonal March Element.

The contributions and prominent features of this study are:

- The novel concept of Generic and Orthogonal March Element enables to perform any suitable combinations of GMEs with algorithm stresses (i.e., address order, addressing scheme, data background and data value).
- A memory test assembly language was introduced that enables the user to write own self-defined memory tests.
- GME MBIST has high flexibility and high fault coverage. Both the linear and non-linear algorithm classes are supported such as March, Galloping/Walking and variations, Butterfly, Hammer and variations, Moving Inversion, Address decoder open/delay related faults and delay test algorithms. In addition, proposed hardware is on-the-field programmable.
- GME MBIST is a low cost and efficient solution for the memory testing. For example, the area overhead is around 9% for a 16K x 16-bit memory and it drops below to 0.01% for a 16M x 16-bit memory. Furthermore, the area of GME MBIST logarithmically increases with the increasing memory sizes. In addition, the variable-length commands and opcodes result in an efficient usage of the Command Memory and further decreases the required area; e.g., it requires only 36 bits to define March C+ algorithm.
- At-speed testing is supported without any complex schemes such as pre-fetching and pipelining. All the information to generate the consecutive memory operations is presented in GMEs. Therefore, there is no cycle loss during a march element.
- Implementation is easily extendable, modifiable and highly modular. Additional new commands and algorithm stresses can be easily added in future. In addition, the user may tailor the hardware depending on the application. This results in a trade-off between the GME MBIST capability, area overhead, maximum frequency

and power consumption. Besides, GME MBIST hardware is generically modeled to shorten the design time-to-market.

- Diagnosis is provided. GME MBIST reports the data and address of the faulty cell out.

9.2 Future Work

Highly modular architecture of GME MBIST with its easily extendable command set simplifies the addition of new features. Here a few suggestions as the future work:

- Operation frequency may be increased by investigating the critical path of the design.
- Proposed GME MBIST may be combined into a BISR system to build a full built-in system (test, diagnosis and repair).
- GME MBIST Processor may be modified to handle multiple memory testing.
- The number of the levels for nested loops may be increased.
- Regarding to the new research on the memory testing, new algorithm stresses and/or new GMEs may be included.

State diagrams of the Controller

A

This section presents the FSM state diagrams of the Controller in detail.

Main states of the Controller

Figure A.1 shows that the Controller has 4 main states: ST_IDLE, ST_START, ST_WAIT_GME_END and ST_DONE. In addition, it consists of many command states. Here, 4 main states will be explained as:

- ST_IDLE: When the system is reseted, GME MBIST enters to this state; and waits to begin the test procedure.
- ST_START: This state decodes the 1^{st} part of the opcode and branches to the valid

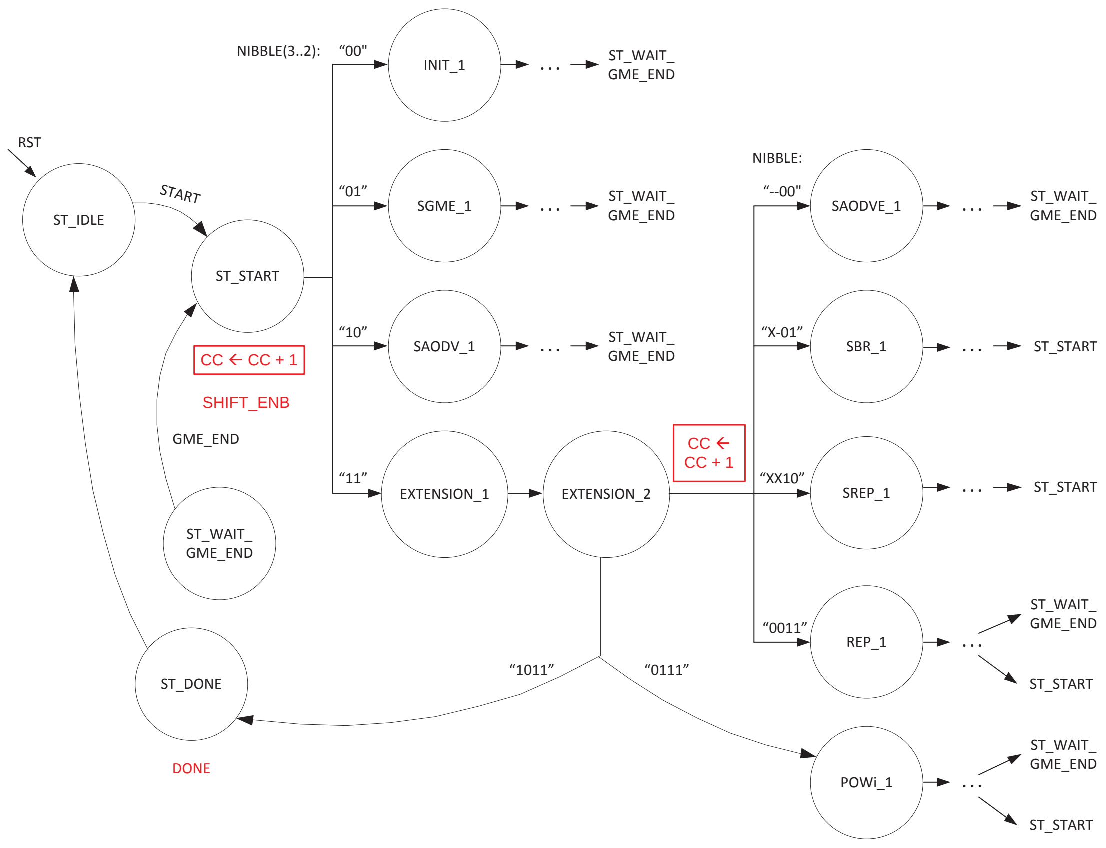


Figure A.1: Main state diagram of the Controller

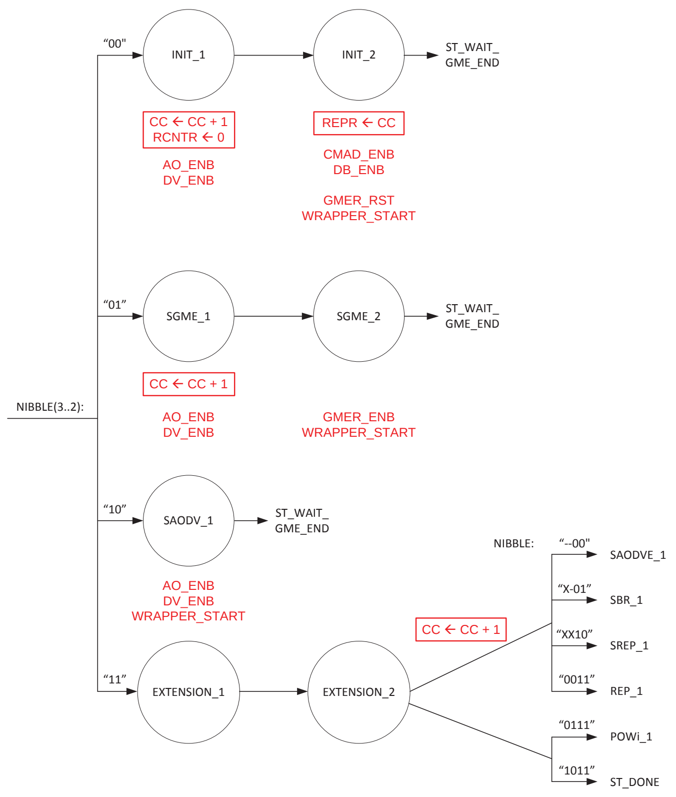


Figure A.2: State diagram for the commands with 2-bit opcode

state. In case of an "INIT", "SGME" or "SAODV" command, it branches into the valid state, otherwise it branches into the EXTENSION_1 state and continues with EXTENSION_2 state. EXTENSION_2 branches into the valid command state by decoding the 2^{nd} and 3^{rd} parts of the opcode.

- ST_WAIT_GME_END: When GME MBIST Wrapper starts to perform a GME to the memory, GME MBIST Processor waits at this state for the GME to be finished.

- ST_DONE: In case of an "END" command, GME MBIST enters to this state to finalize the test flow.

Since the GME MBIST commands have the variable-length opcodes, commands can be classified depending on their opcode lengths: commands with 2-bit, 4-bit and 6-bit opcode. Command opcodes are shown in Figure 7.4. Here, they will be discussed.

State diagram for the commands with 2-bit opcode

Figure A.2 shows the state diagram for the commands with 2-bit opcode. When Opc (i.e., *NIBBLE[3..2]*) is "00", ST_START state branches to INIT_1 state; "01" to SGME_1 state; "10" to SAODV_1 state. Then certain command operations (e.g., reseting/loading registers) are done. At the final states of each command, GME MBIST Wrapper is

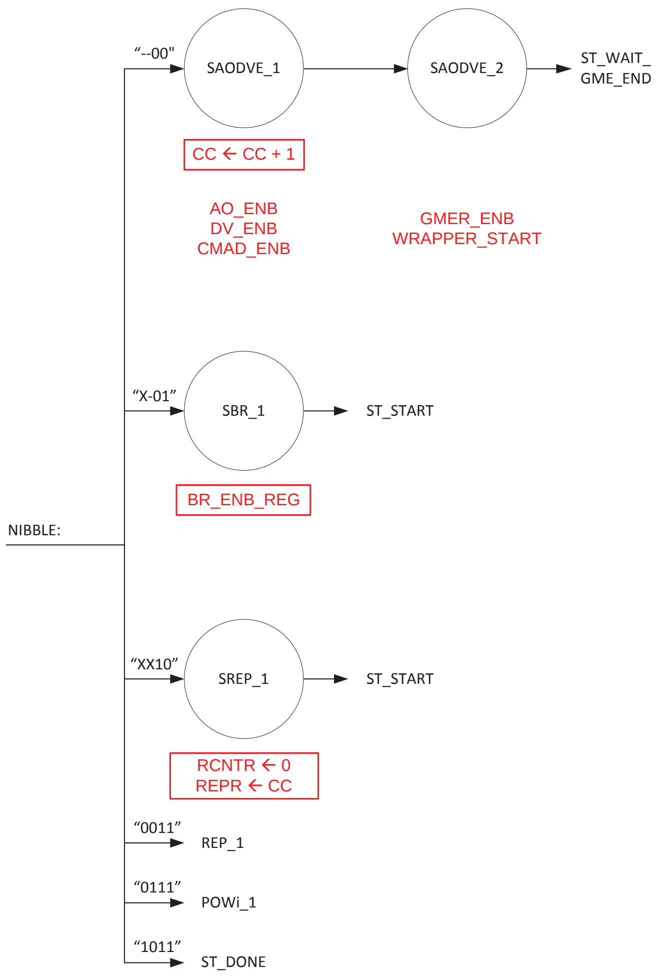


Figure A.3: State diagram for the commands with 4-bit opcode

activated by the "WRAPPER_START" signal. Afterwards, GME MBIST Processor enters to ST_WAIT_GME_END state and waits to receive "GME_END" signal from Memory Wrapper.

State diagram for the commands with 4-bit opcode

In case of an Opc "11", GME MBIST Controller realizes that the command opcode is longer than 2-bit. For the opcode's 2nd part, one more word is fetched in EXTENSION_1 state, and is decoded in EXTENSION_2 state, as shown in Figure A.2. Figure A.3 shows the state diagram for the commands with 4-bit opcode. When the 2nd part of the opcode (*NIBBLE[1..0]*) is "00", EXTENSION_2 branches to SAODVE_1 state; "01" to SBR_1 state; "10" to SREP_1 state. Afterwards, certain command operations are done. At the final states, SAODVE_2 further proceeds to ST_WAIT_GME_END state; whereas SBR_1 and SREP_1 return to ST_START state to fetch a new command. Reason is that they do not perform any GME operations on the memory (i.e., they only select GME bank

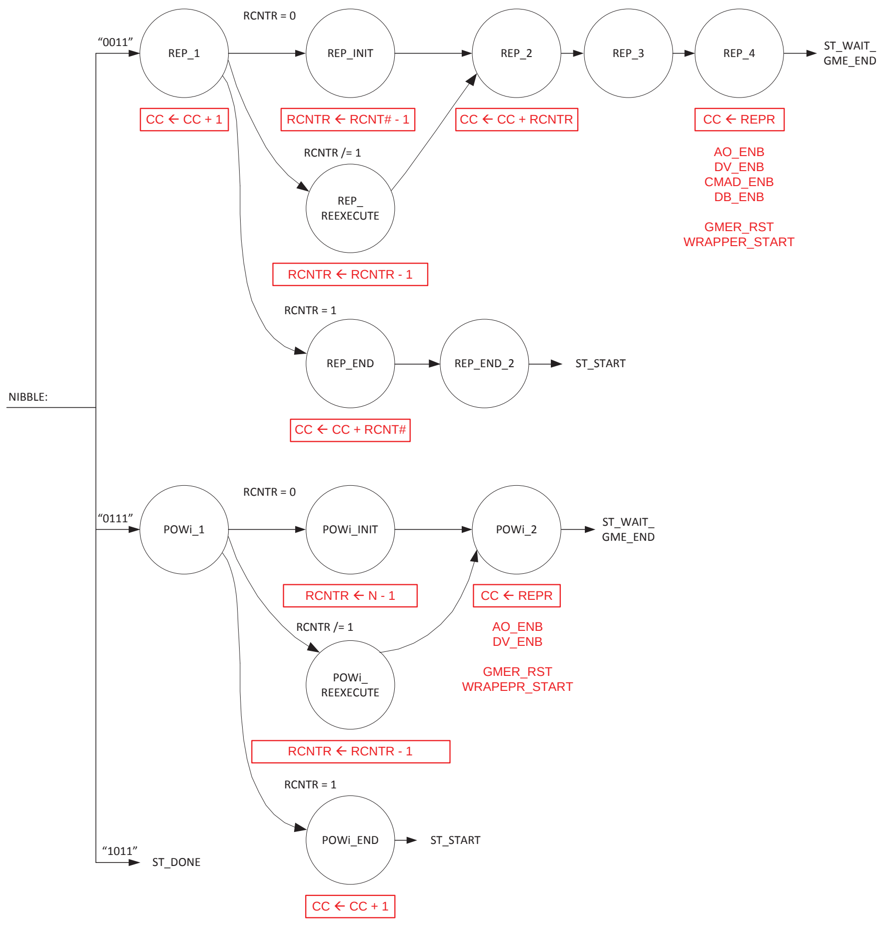


Figure A.4: State diagram for the commands with 6-bit opcode

or set the *Re-execute Entry Point Register (REPR)*).

State diagram for the commands with 6-bit opcode

When the 2nd part of the opcode (*NIBBLE[1..0]*) is "11", the 3rd part of the opcode (*NIBBLE[3..2]*) is checked. If it is "00", EXTENSION_2 branches to REP_1 state; "01" to POWi_1 state; "10" to ST_DONE state as shown in Figure A.4. REP_1 or POWi_1 states branch to the REP/POWi initialization, re-execution or end states depending on the *Re-execute CouNT Register (RCNTR)* value. At the initialization states, RCNTR is loaded by RCNT#-1 for REP command; by N-1 for POWi command. At the re-execution states, RCNTR is decremented by 1. After both the initialization and re-execution states, FSM further proceeds to the REP_2 or POWi_2 state, and certain command operations are done (e.g., reseting/loading registers). Afterwards, GME MBIST Processor enters to ST_WAIT_GME_END state and waits for the Memory Wrapper to finish the specified GME. At the end states, CC pointer jumps to the next command after the REP/POWi commands, and system returns back to ST_START state to fetch this new command.

State diagram of the Address, Data & Control Generators B

Figure B.1 shows the FSM for the Address, Data and Control Generators. Here, it is discussed.

- ST_IDLE: When the system is reseted, GME MBIST Wrapper waits to begin the test procedure.
- ST_LOAD: Beginning and end addresses for inner and outer loop address counters are loaded. Afterwards system branches to the specified GME state by BGMER.
- GME States: GMEs have varying number of memory operations. For example in Table 6.2, B(GME): 0(0), $\Updownarrow$ (wD), has only one write operation; B(GME): 0(3), $\Updownarrow$ $(rD, w\overline{D}, r\overline{D})$, has totally three: one write and two read operations; B(GME): 1(2), $\Uparrow_v$ $(wD_v, \Uparrow_{-v}$ $(r\overline{D}, rD_v), w\overline{D}_v)$, has totally four: 2 write and 2 read operations. Therefore, B(GME): 0(0) has one state; B(GME): 0(3) has three states; B(GME): 1(2) has four states.
- GME_END: When performing a specified GME finishes, GME MBIST Wrapper enters to GME_END state, and informs the GME MBIST Processor. Afterwards, it returns back to ST_IDLE state.

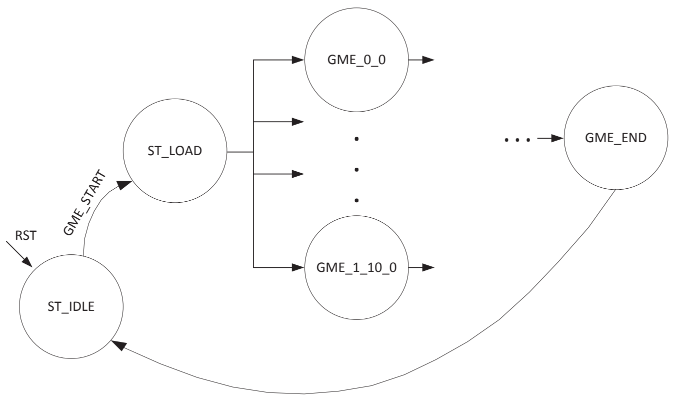


Figure B.1: State diagram for the Address, Data and Control Generators

VHDL coding for the Address, Data & Control Generators

C

Here, VHDL code examples are given for the implementation of the Address generator.

Listing C.1: VHDL coding of B(GME): 0(0) Address Generator

```
    when GME_0_0 =>
        if ADDR_OUT_REG /= END_ADDR_V_REG then
           STATE <= GME_0_0;
           ADDER_CONTROL <= "010";
        else
           STATE <= GME_END;
           ADDER_CONTROL <= "011";
        end if;
        BASE_LOAD <= "00";
```

Listing C.2: VHDL coding of B(GME): 0(3) Address Generator

```
    when GME_0_3_0 =>
        STATE <= GME_0_3_1;
        BASE_LOAD <= "00";
        ADDER_CONTROL <= "000";
    when GME_0_3_1 =>
        STATE <= GME_0_3_2;
        BASE_LOAD <= "00";
        ADDER_CONTROL <= "000";
    when GME_0_3_2 =>
        if ADDR_OUT_REG /= END_ADDR_V_REG then
           STATE <= GME_0_3_0;
           ADDER_CONTROL <= "010";
        else
           STATE <= GME_END;
           ADDER_CONTROL <= "011";
        end if;
        BASE_LOAD <= "00";
```

Listing C.3: VHDL coding of B(GME): 1(2) Address Generator

```
    when GME_1_2_0 =>
        STATE <= GME_1_2_1;
        BASE_LOAD <= "10";
        ADDER_CONTROL <= "011";
    when GME_1_2_1 =>
        if ADDR_OUT_REG /= BASE_ADDR_AFTER then
            STATE <= GME_1_2_2;
            BASE_LOAD <= "00";
            ADDER_CONTROL <= "000";
        elsif ADDR_OUT_REG /= END_ADDR_INNER_REG then
            STATE <= GME_1_2_1;
            BASE_LOAD <= "00";
            ADDER_CONTROL <= "010";
        else
           STATE <= GME_1_2_3;
           BASE_LOAD <= "10";
           ADDER_CONTROL <= "000";
        end if;
    when GME_1_2_2 =>
        if ADDR_OUT_REG /= END_ADDR_INNER_REG then
            STATE <= GME_1_2_1;
            ADDER_CONTROL <= "010";
        else
            STATE <= GME_1_2_3;
            ADDER_CONTROL <= "001";
        end if;
        BASE_LOAD <= "00";
    when GME_1_2_3 =>
        if ADDR_OUT_REG /= END_ADDR_V_REG then
           STATE <= GME_1_2_0;
           ADDER_CONTROL <= "010";
        else
            STATE <= GME_END;
            ADDER_CONTROL <= "011";
        end if;
        BASE_LOAD <= "00";
```

B(GME): 0(0), 0(3) and 1(2) states from the Table 6.2 are chosen in those examples. Listing C.1, C.2 and C.3 show that how a state further proceeds to the next state. In addition, they show the control signals for the up-only counter and the base cell register.

Listing C.4: VHDL codings of B(GME): 0(0)-0(3)-1(2) Data & Control Generators

```
    when GME_0_0 =>
        Data_V(DB_IN,DV_IN,ADDR_OUT_REG_IN,DATA);
        Read_Write('1',N_CE,N_WE,N_OE);
    ------------------------------------------------------------------
    when GME_0_3_0 =>
        Data_V(DB_IN,DV_IN,ADDR_OUT_REG_IN,DATA);
        Read_Write('0',N_CE,N_WE,N_OE);
    when GME_0_3_1 =>
```

```
          Data_V(DB_IN,NOT DV_IN,ADDR_OUT_REG_IN,DATA);
          Read_Write('1',N_CE,N_WE,N_OE);
     when GME_0_3_2 =>
          Data_V(DB_IN,NOT DV_IN,ADDR_OUT_REG_IN,DATA);
          Read_Write('0',N_CE,N_WE,N_OE);
------------------------------------------------------
     when GME_1_2_0 =>
          Data_V(DB_IN,DV_IN,ADDR_OUT_REG_IN,DATA);
          Read_Write('1',N_CE,N_WE,N_OE);
     when GME_1_2_1 =>
          Data_V(DB_IN,NOT DV_IN,ADDR_OUT_REG_IN,DATA);
          if ADDR_OUT_REG_IN /= BASE_ADDR_AFTER_IN then
               Read_Write('0',N_CE,N_WE,N_OE);
          else
               N_CE <= '1';
               N_WE <= '1';
               N_OE <= '1';
          end if;
     when GME_1_2_2 =>
          Data_V(DB_IN,DV_IN,ADDR_OUT_REG_IN,DATA);
          Read_Write('0',N_CE,N_WE,N_OE);
     when GME_1_2_3 =>
          Data_V(DB_IN,NOT DV_IN,ADDR_OUT_REG_IN,DATA);
          Read_Write('1',N_CE,N_WE,N_OE);
```

Here, VHDL code examples are given for the implementation of the Data & Control Generator. B(GME): 0(0), 0(3) and 1(2) states from the Table 6.2 are chosen in those examples. Listing C.4 shows that the data and control signals are generated by the procedures.

Listing C.5: VHDL coding of data generation procedure

```
if DB = "00" then                 --sDB
     DATA_OUT <= (others => DV);
elsif DB = "01" then              --bDB
     DATA_OUT <= (others => DV XOR ROW_ADDR(0) XOR COL_ADDR(0));
elsif DB = "10" then              --rDB
     DATA_OUT <= (others => DV XOR ROW_ADDR(0));
else                              --cDB
     DATA_OUT <= (others => DV XOR COL_ADDR(0));
end if;
```

Listing C.6: VHDL coding of control generation procedure

```
     if R_W = '0' then --Read Operation
          N_CE_OUT <= '0';
          N_WE_OUT <= '1';
```

```
        N_OE_OUT <= '0';
    else                --Write Operation
        N_CE_OUT <= '0';
        N_WE_OUT <= '0';
        N_OE_OUT <= '1';
    end if;
```

Here, VHDL code examples are given for the the data and control procedures. Listing C.5 shows that the data is simply generated by AND/XORing the certain row and column address bits. Listing C.6 shows that the control signals (i.e., chip, write and output enables) are generated depending on the read/write operations.

Bibliography

[1] M. S. Abadir and H. K. Reghbati, *Functional Testing of Semiconductor Random Access Memories*, ACM Comput. Surv. **15** (1983), no. 3, 175–198.

[2] R. D. Adams, *High Performance Memory Testing: Design Principles, Fault Modeling and Selft-Test*, Kluwer Academic Pub., Dordrecht, NL, 2003.

[3] R. D. Adams and E. S. Cooley, *False write through and un-restored write electrical level fault models for SRAMs*, Proceedings of the International Workshop on Memory Technology, Design and Testing, Aug 1997, pp. 27–32.

[4] V.D. Agrawal, C.R. Kime, and K.K. Saluja, *A tutorial on built-in self-test. I. Principles*, IEEE Design and Test of Computers **10** (1993), no. 1, 73–82.

[5] K. Akiyama and K.K. Saluja, *A method of reducing aliasing in a built-in self-test environment* , IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **10** (1991), no. 4, 548–553.

[6] Z. Al-Ars and Ad.J. van de Goor, *Approximating infinite dynamic behavior for DRAM cell defects*, VTS '02: Proceedings 20th IEEE VLSI Test Symposium, 2002, pp. 401–406.

[7] Zaid Al-Ars, Said Hamdioui, Georgi Gaydadjiev, and Stamatis Vassiliadis, *Test set development for cache memory in modern microprocessors*, IEEE Trans. Very Large Scale Integr. Syst. **16** (2008), no. 6, 725–732.

[8] A. Allan, D. Edenfeld, Jr. Joyner, W.H., A.B. Kahng, M. Rodgers, and Y. Zorian, *2001 Technology Roadmap for Semiconductors*, Computer **35** (ITRS 2002), no. 1, 42–53.

[9] D. Appello, P. Bernardi, A. Fudoli, M. Rebaudengo, M. Sonza Reorda, V. Tancorre, and M. Violante, *Exploiting Programmable BIST For The Diagnosis of Embedded Memory Cores*, Test Conference, International **0** (2003), 379.

[10] V. Arora, W.B. Jone, D.C. Huang, and S.R. Das, *A parallel built-in self-diagnostic method for nontraditional faults of embedded memory arrays*, IEEE Transactions on Instrumentation and Measurement **53** (2004), no. 4, 915–932.

[11] M. Azimane and A. K. Majhi, *New test methodology for resistive open defect detection in memory address decoders*, Proceedings of the 22nd IEEE VLSI Test Symposium, April 2004, pp. 123–128.

[12] S. Banerjee and D.R. Chowdhury, *Built-in self-test for flash memory embedded in SoC*, DELTA '06: Third IEEE International Workshop on Electronic Design, Test and Applications, Jan. 2006, pp. 6 pp.–384.

[13] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, and P. Prinetto, *Automatic March tests generation for static and dynamic faults in SRAMs*, European Test Symposium, IEEE Computer Society, May 2005, pp. 122–127.

[14] ______, *March AB, March AB1: new March tests for unlinked dynamic memory faults*, ITC '05: Proceedings of IEEE International Test Conference, Nov. 2005, pp. 8 pp.–841.

[15] A. Benso, S. Chiusano, G. Di Natale, and P. Prinetto, *An on-line BIST RAM architecture with self-repair capabilities*, IEEE Transactions on Reliabilit **51** (2002), no. 1, 123–128.

[16] A. Bommireddy, J. Khare, S. Shaikh, and S.-T. Su, *Test and debug of networking SoCs-a case study*, Proceedings of 18th IEEE VLSI Test Symposium, 2000, pp. 121–126.

[17] S. Boutobza, M. Nicolaidis, K.M. Lamara, and A. Costa, *Programmable memory BIST*, IEEE International Test Conference Proceedings, ITC'05, 2005, pp. 10 pp.–1164.

[18] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Inc., 1976.

[19] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, Frontiers in electronic testing, Springer Publishing Company, Incorporated, New York, NY, USA, 2000.

[20] O. Caty, I. Bayraktaroglu, A. Majumdar, R. Lee, J. Bell, and L. Curhan, *Instruction based bist for board/system level test of external memories and internconnects*, ITC '03: Proceeding of International Test Conference, vol. 1, 30-Oct. 2, 2003, pp. 961–970.

[21] H. Cheung and S.K. Gupta, *A BIST methodology for comprehensive testing of RAM with reduced heat dissipation*, ITC '96: Proceedings of International Test Conference, Oct 1996, pp. 386–395.

[22] Faraday Technology Corp., *Fsd0a a sh 90 nm synchronous high density single-port sram compiler*, October 2006.

[23] J. H. de Jonge and A. J. Smeulders, *Moving inversions test pattern is thorough, yet speedy*, Comput. Des. (1976), 169–173.

[24] R. Dekker, F. Beenker, and L. Thijssen, *Fault modeling and test algorithm development for static random access memories*, Test Conference, 1988. Proceedings. New Frontiers in Testing, International, Sep. 1988, pp. 343–352.

[25] ______, *A realistic fault model and test algorithms for static random access memories*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **9** (1990), no. 6, 567–572.

[26] Luigi Dilillo, Patrick Girard, Serge Pravossoudovitch, Arnaud Virazel, Simone Borri, and Magali Hage-Hassan, *ADOFs and Resistive-ADOFs in SRAM Address Decoders: Test Conditions and March Solutions*, J. Electron. Test. **22** (2006), no. 3, 287–296.

[27] J. Dreibelbis, J. Barth, H. Kalter, and R. Kho, *Processor-based built-in self-test for embedded DRAM*, IEEE Journal of Solid-State Circuits **33** (1998), no. 11, 1731–1740.

[28] Xiaogang Du, N. Mukherjee, Wu-Tung Cheng, and S.M. Reddy, *Full-speed field-programmable memory BIST architecture*, ITC '05: Proceedings of IEEE International Test Conference, Nov. 2005, pp. 9 pp.–1173.

[29] Xiaogang Du, Nilanjan Mukherjee, Wu-Tung Cheng, and Sudhakar M. Reddy, *Full-Speed Field Programmable Memory BIST Supporting Multi-level Looping*, MTDT '05: Proceedings of the IEEE International Workshop on Memory Technology, Design, and Testing, 2005, pp. 67–71.

[30] Xiaogang Du, Nilanjan Mukherjee, Chris Hill, Wu-Tung Cheng, and Sudhakar M. Reddy, *A Field Programmable Memory BIST Architecture Supporting Algorithms with Multiple Nested Loops*, ATS '06: Proceedings of the 15th Asian Test Symposium, 2006, pp. 287–292.

[31] M.S. Elsaholy, S.I. Shaheen, and R.H. Seireg, *A unified analytical expression for aliasing error probability using single-input external- and internal-XOR LFSR*, IEEE Transactions on Computers **47** (1998), no. 12, 1414–1417.

[32] Bai Hong Fang and N. Nicolici, *Power-constrained embedded memory BIST architecture*, Proceedings of 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Nov 2003, pp. 451–458.

[33] M. Franklin and K.K. Saluja, *Built-in self-testing of random-access memories*, Computer **23** (1990), no. 10, 45–56.

[34] I. A. Grout, *Integrated Circuit Test Engineering: Modern Techniques*, Springer Science + Business Media B. V., 2006.

[35] S.K. Gupta and D.K. Pradhan, *A new framework for designing and analyzing BIST techniques: computation of exact aliasing probability*, Proceedings, New Frontiers in Testing, International Test Conference, Sep 1988, pp. 329–342.

[36] S. Hamdioui, *Testing Multi-Port Memories: Theory and Practice*, Ph.D. thesis, Delft University of Technology, January 2001.

[37] ______, *Testing Static Random Access Memories: Defects, Fault Models and Test Patterns*, Kluwer Academic Pub., 2004.

[38] S. Hamdioui and Z. Al-Ars, *Scan More with Memory Scan Test*, IEEE Proc. of International Conference on Design and Technology of Integrated Systems in Nanoera, April 2009, pp. 204–209.

[39] S. Hamdioui, Z. Al-Ars, G. N. Gaydadjiev, and J. D. Reyes, *Comparison of Static and Dynamic Faults in 65nm Memory Technology*, IDT '06: Proceedings of the 1st IEEE International Design and Test Workshop, November 2006.

[40] ———, *Investigation of Single-Cell Dynamic Faults in Deep-Submicron Memory Technologies*, ETS '06: Proceedings of the 11th IEEE European Test Symposium Digest of Papers, May 2006, pp. 21–25.

[41] S. Hamdioui, Z. Al-Ars, J. Jimenez, and J. Calero, *PPM Reduction on Embedded Memories in System on Chip*, IEEE proceedings of European Test Symposium, May 2007, pp. 85–90.

[42] S. Hamdioui, Z. Al-Ars, and A. J. van de Goor, *Testing Static and Dynamic Faults in Random Access Memories*, IEEE VLSI Test Symposium, January 2002, pp. 395–400.

[43] S. Hamdioui, Z. Al-Ars, A. J. van de Goor, and M. Rodgers, *Linked faults in random access memories: concept, fault models, test algorithms, and industrial results*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23** (2004), no. 5, 737–757.

[44] S. Hamdioui, Z. Al-Ars, and A.J. van de Goor, *Opens and Delay Faults in CMOS RAM Address Decoders*, IEEE Transactions on Computers **55** (2006), no. 12, 1630–1639.

[45] S. Hamdioui, Z. Al-Ars, A.J. van de Goor, and M. Rodgers, *March SL: a test for all static linked memory faults*, ATS '03: Proceedings of the 12th Asian Test Symposium, IEEE Computer Society, Nov. 2003, pp. 372–377.

[46] S. Hamdioui, G. N. Gaydadjiev, and A. J. van de Goor, *A Fault Primitive Based Analysis of Dynamic Memory Faults*, Proceedings of PRORISC '03, November 2003, pp. 84–89.

[47] S. Hamdioui, A. J. van de Goor, and M. Rodgers, *March SS: A Test for All Static Simple RAM Faults*, IEEE International Workshop on Memory Technology, Design and Testing (2002), 95.

[48] S. Hamdioui and A.J. Van De Goor, *An experimental analysis of spot defects in SRAMs: realistic fault models and tests*, ATS '00: Proceedings of the 9th Asian Test Symposium, 2000, pp. 131–138.

[49] S. Hamdioui, A.J. van de Goor, J.D Reyes, and M. Rodgers, *Memory test experiment: industrial results and data*, IEE Proceedings of Computers and Digital Techniques (2006), 1–8.

[50] S. Hamdioui, R. Wadsworth, J. Delos Reyes, and A.J. van de Goor, *Importance of dynamic faults for new SRAM technologies*, Proceedings of the 8th IEEE European Test Workshop, May 2003, pp. 29–34.

[51] Said Hamdioui, Zaid Al-Ars, Ad J. Van De Goor, and Mike Rodgers, *Dynamic Faults in Random-Access-Memories: Concept, Fault Models and Tests*, Journal of Electronic Testing **19** (2003), no. 2, 195–205.

[52] Said Hamdioui, Georgi Gaydadjiev, and Ad J. van de Goor, *The State-of-Art and Future Trends in Testing Embedded Memories*, IEEE International Workshop on Memory Technology, Design and Testing (2004), 54–59.

[53] G. Harutunyan, V. A. Vardanian, and Y. Zorian, *Minimal March Tests for Unlinked Static Faults in Random Access Memories*, VTS '05: Proceedings of the 23rd IEEE VLSI Test Symposium, IEEE Computer Society, 2005, pp. 53–59.

[54] ———, *Minimal March Tests for Dynamic Faults in Random Access Memories*, ETS '06: Proceedings of the Eleventh IEEE European Test Symposium, IEEE Computer Society, 2006, pp. 43–48.

[55] G. Harutunyan, V. A. Vardanian, and Y. Zorian Zorian, *Minimal March Test Algorithm for Detection of Linked Static Faults in Random Access Memories*, TS '06: Proceedings of the 24th IEEE VLSI Test Symposium, IEEE Computer Society, 2006, pp. 120–127.

[56] Chih-Tsun Huang, Jing-Reng Huang, Chi-Feng Wu, Cheng-Wen Wu, and Tsin-Yuan Chang, *A programmable BIST core for embedded DRAM*, IEEE Design and Test of Computers **16** (1999), no. 1, 59–70.

[57] Shi-Yu Huang and Ding-Ming Kwai, *A high-speed built-in-self-test design for DRAMs*, International Symposium on VLSI Technology, Systems, and Applications, 1999, pp. 50–53.

[58] Synopsys Inc., *Design compiler 2010*, February 2010.

[59] A. Ivanov, B.K. Tsuji, and Y. Zorian, *Programmable BIST space compactors*, IEEE Transactions on Computers **45** (1996), no. 12, 1393–1404.

[60] N.T. Jarwala and D.K. Pradhan, *TRAM: a design methodology for high-performance, easily testable, multimegabit RAMs*, IEEE Transactions on Computers **37** (1988), no. 10, 1235–1250.

[61] R. V. Joshi, R. Q. Williams, E. Nowak, K. Kim, J. Beintner, T. Ludwig, I. Aller, and C. Chuang, *FinFET SRAM for High-Performance Low-Power Applications*, Tech. report, IBM, 2004.

[62] T. Kameda, S. Pilarski, and A. Ivanov, *Notes on multiple input signature analysis*, IEEE Transactions on Computers **42** (1993), no. 2, 228–234.

[63] H.C. Kim, H.-S. Jun, Xinli Gu, and S.S. Chung, *At-speed interconnect test and diagnosis of external memories on a system*, ITC '04: Proceedings of International Test Conference, Oct. 2004, pp. 156–162.

[64] Ilyoung Kim, Y. Zorian, G. Komoriya, H. Pham, F.P. Higgins, and J.L. Lewandowski, *Built in self repair for embedded high density SRAM*, Proceedings of International Test Conference, Oct 1998, pp. 1112–1119.

[65] M. Klaus and A. J. Van de Goor, *Tests for resistive and capacitive defects in address decoders*, Proceedings of the 10th Asian Test Symposium, 2001, pp. 31–36.

[66] Jr. Knaizuk, J. and C. R. P. Hartmann, *An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories*, IEEE Trans. Comput. **26** (1977), no. 11, 1141–1144.

[67] S.-Y. Kuo and W.K. Fuchs, *Modelling and algorithms for spare allocation in reconfigurable VLSI*, Computers and Digital Techniques, IEEE Proceedings **139** (1992), no. 4, 323–328.

[68] Sy-Yen Kuo and W.K. Fuchs, *Efficient Spare Allocation for Reconfigurable Arrays*, Design and Test of Computers, IEEE **4** (1987), no. 1, 24–31.

[69] Y. Leblebici and S. M. Kang, *CMOS Digital Integrated Circuits: Analysis and Design*, $3^r d$ ed., McGraw-Hill, 2003.

[70] K. J. Lee, J. Y. Wu, and W. B. Jone, *Built-in Self-Test for Multiple Memories in a chip*, US Patent 6360342 B1, Mar 2002.

[71] M. Lobetti Bodoni, A. Benso, S. Chiusano, S. Di Carlo, G. Di Natale, and P. Prinetto, *An effective distributed BIST architecture for RAMs*, Proceedings of IEEE European Test Workshop, 2000, pp. 119–124.

[72] W. Maly, *Modeling of Lithography Related Yield Losses for CAD of VLSI Circuits*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **4** (1985), no. 3, 166–177.

[73] Marian Marinescu, *Simple and Efficient Algorithms for Functional RAM Testing*, ITC, 1982, pp. 236–239.

[74] P. Mazumder, *Parallel testing of parametric faults in a three-dimensional dynamic random-access memory*, IEEE Journal of Solid-State Circuits **23** (1988), no. 4, 933–941.

[75] M.G. Mohammad and K.K. Saluja, *Flash memory disturbances: modeling and test*, VTS '01: Proceedings of 19th IEEE VLSI Test Symposium, 2001, pp. 218–224.

[76] B. Nadeau-Dostie, A. Silburt, and V.K. Agarwal, *Serial interfacing for embedded-memory testing*, IEEE Design and Test of Computers **7** (1990), no. 2, 52–63.

[77] ______, *Serial testing Technique for Embedded Memories*, US Patent 4969148, Nov 1990.

[78] Benoit Nadeau-Dostie, *Design for at-speed test, diagnosis and measurement*, Kluwer Academic Publishers, May 2002.

[79] R. Nair, *Comments on "An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories"*, IEEE Trans. Comput. **28** (1979), no. 3, 258–261.

[80] R. Nair, S. M. Thatte, and J. A. Abraham, *Efficient Algorithms for Testing Semiconductor Random-Access Memories*, IEEE Trans. Comput. **27** (1978), no. 6, 572–576.

[81] J. Otterstedt, D. Niggemeyer, and T. W. Williams, *Detection of CMOS address decoder open faults with March and pseudo random memory tests*, Proceedings of InternationalTest Conference, Oct 1998, pp. 53–62.

[82] Youngkyu Park, Jaeseok Park, Tewoo Han, and Sungho Kang, *An Effective Programmable Memory BIST for Embedded Memory*, IEICE Transactions on Information and Systems **E92-D** (2009), no. 12, 2508–2511.

[83] Andrei Pavlov and Manoj Sachdev, *CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies*, Springer Science + Business Media B.V., New York, NY, USA, 2008.

[84] L. J. Popyack, *Micro-coded Built-in Self-test Apparatus for a Memory Array*, US Patents 5224101, June 1993.

[85] T.J. Powell, Wu-Tung Cheng, J. Rayhawk, O. Samman, P. Policke, and S. Lai, *Bist for deep submicron asic memories with high performance application*, ITC '03: Proceedings of the International Test Conference, Oct 2003, pp. 386–392.

[86] T.J. Powell, F. Hii, and D. Cline, *A 256 Meg SDRAM BIST for disturb test application*, Proceedings of International Test Conference (1997), 200–208.

[87] D.K. Pradhan, S.K. Gupta, and M.G. Karpovsky, *Aliasing probability for multiple input signature analyzer*, IEEE Transactions on Computers **39** (1990), no. 4, 586–591.

[88] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2^{nd} ed., Prentice Hall Electronics and VLSI Series, Pearson Education, Inc., 2003.

[89] S.M. Reddy, K.K. Saluja, and M.G. Karpovsky, *A data compression technique for built-in self-test*, IEEE Transactions on Computers **37** (1988), no. 9, 1151–1156.

[90] M. Sachdev, *Open defects in CMOS RAM address decoders*, IEEE Design and Test of Computers **14** (1997), no. 2, 26–33.

[91] N.R. Saxena and J.P. Robinson, *Syndrome and transition count are uncorrelated*, IEEE Transactions on Information Theory **34** (1988), no. 1, 64–69.

[92] Hitachi Semiconductors, *Hitachi HM628128B Series 1 Mb SRAM*, 1997.

[93] Chin-Lung Su, Rei-Fu Huang, and Cheng-Wen Wu, *A processor-based built-in self-repair design for embedded memories*, ATS '03: 12th Asian Test Symposium, Nov 2003, pp. 366–371.

[94] D. S. Suk and S. M. Reddy, *A March Test for Functional Faults in Semiconductor Random Access Memories*, IEEE Trans. Comput. **30** (1981), no. 12, 982–985.

[95] T. Takeshima, M. Takada, H. Koike, H. Watanabe, S. Koshimaru, K. Mitake, W. Kikuchi, T. Tanigawa, T. Murotani, K. Noda, K. Tasaka, K. Yamanaka, and K. Koyama, *A 55-ns 16-Mb DRAM with built-in self-test function using microprogram ROM*, IEEE Journal of Solid-State Circuits **25** (1990), no. 4, 903–911.

[96] Luigi Ternullo, Jr., R. Dean Adams, John Connor, and Garret S. Koch, *Deterministic Self-Test of a High-Speed Embedded Memory and Logic Processor Subsystem*, Proceedings of the IEEE International Test Conference on Driving Down the Cost of Test (Washington, DC, USA), IEEE Computer Society, 1995, pp. 33–44.

[97] Ching-Hong Tsai and Cheng-Wen Wu, *Processor-Programmable Memory BIST for Bus-Connected Embedded Memories*, ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference (New York, NY, USA), ACM, 2001, pp. 325–330.

[98] A.J. van de Goer and J. de Neef, *Industrial evaluation of DRAM tests*, DATE '99: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, 1999, pp. 623–630.

[99] A. J. van de Goor, *Using March Tests to Test SRAMs*, IEEE Design and Test of Computers **10** (1993), no. 1, 8–14.

[100] ———, *Testing Semiconductor Memories: Theory and Practice*, Verzijl, Gouda, NL, 1998.

[101] A. J. van de Goor and G. N. Gaydadjiev, *An analysis of (linked) address decoder faults*, MTDT '97: Proceedings of the 1997 IEEE International Workshop on Memory Technology, Design and Testing, IEEE Computer Society, 1997, p. 13.

[102] ———, *March U: a test for unlinked memory faults*, IEEE Proceedings on Circuits, Devices and Systems **144** (1997), no. 3, 155–160.

[103] A. J. van de Goor, G. N. Gaydadjiev, V. G. Mikitjuk, and V. N. Yarmolik, *March LR: a test for realistic linked faults*, VTS '96: Proceedings of the 14th IEEE VLSI Test Symposium, IEEE Computer Society, 1996, p. 272.

[104] A. J. van de Goor, G. N. Gaydadjiev, V. N. Yarmolik, and V. G. Mikitjuk, *March LA: a test for linked memory faults*, EDTC '97: Proceedings of the 1997 European conference on Design and Test, IEEE Computer Society, 1997, p. 627.

[105] Ad J. van de Goor and Zaid Al-Ars, *Functional Memory Faults: a Formal Notation and a Taxonomy*, VTS '00: Proceedings of the 18th IEEE VLSI Test Symposium (Washington, DC, USA), IEEE Computer Society, 2000, p. 281.

[106] Ad J. van de Goor and Alexander Paalvast, *Industrial Evaluation of DRAM SIMM Tests*, ITC '00: Proceedings of the 2000 IEEE International Test Conference, IEEE Computer Society, 2000, p. 426.

[107] Ad.J. van de Goor, S. Hamdioui, and R. Wadsworth, *Detecting faults in the peripheral circuits and an evaluation of SRAM tests*, ITC '04: Proceedings of International Test Conference, Oct. 2004, pp. 114–123.

[108] A.J. van de Goor, S. Hamdioui, G.N. Gaydadjiev, and Z. Al-Ars, *New algorithms for address decoder delay faults and bit line imbalance faults*, Asian Test Symposium, 2009. ATS '09., Nov 2009, pp. 391 –396.

[109] A.J. van de Goor and J.E. Simonse, *Defining SRAM resistive defects and their simulation stimuli*, ATS '09: Proceedings of the 8th AsianTest Symposium, 1999, pp. 33–40.

[110] B. Wang, Y. Wu, and A. Ivanov, *Designs for reducing test time of distributed small embedded SRAMs*, DFT '04:Proceedings of 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Oct 2004, pp. 120–128.

[111] Chih-Wea Wang, Ruey-Shing Tzeng, Chi-Feng Wu, Chih-Tsun Huang, Cheng-Wen Wu, Shi-Yu Huang, Shyh-Horng Lin, and Hsin-Po Wang, *A built-in self-test and self-diagnosis scheme for heterogeneous SRAM clusters*, Proceedings of 10th Asian Test Symposium, 2001, pp. 103–108.

[112] Wei-Lun Wang, Kuen-Jong Lee, and Jhing-Fa Wang, *An on-chip march pattern generator for testing embedded memory cores*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9** (2001), no. 5, 730–735.

[113] T.W. Williams and W. Daehn, *Aliasing probability for multiple input signature analyzers with dependent inputs*, CompEuro '89: Proceedings of 'VLSI and Computer Peripherals. VLSI and Microelectronic Applications in Intelligent Peripherals and their Interconnection Networks', May 1989, pp. 5/120–5/127.

[114] K. Yamasaki, I. Suzuki, A. Kobayashi, K. Horie, Y. Kobayashi, H. Aoki, H. Hayashi, K. Tada, K. Tsutsumida, and K. Higeta, *External memory BIST for system-in-package*, ITC '05: Proceedings of IEEE International Test Conference, Nov. 2005, pp. 10 pp.–1154.

[115] Jen-Chieh Yeh, Chi-Feng Wu, Kuo-Liang Cheng, Yung-Fa Chou, Chih-Tsun Huang, and Cheng-Wen Wu, *Flash memory built-in self-test using March-like algorithms*, Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications, 2002, pp. 137–141.

[116] Dongkyu Youn, Taehyung Kim, and Sungju Park, *A microcode-based memory BIST implementing modified march algorithm* , Proceedings of 10th Asian Test Symposium, 2001, pp. 391–395.

[117] Kamran Zarrineh and Shambhu J. Upadhyaya, *A New Framework For Automatic Generation, Insertion and Verification of Memory Built-In Self Test Units*, 17th IEEE VLSI Test Symposium (1999), 391–396.

[118] Y. Zorian, *A structured approach to macrocell testing using built-in self-test* , Proceedings of the IEEE Custom Integrated Circuits Conference, May 1990, pp. 28.3/1–28.3/4.

[119] ______, *A distributed BIST control scheme for complex VLSI devices*, Proceedings of Eleventh Annual IEEE VLSI Test Symposium Digest of Papers, Apr. 1993, pp. 4–9.

[120] ______, *Embedded memory test and repair: infrastructure IP for SOC yield*, Proceedings of International Test Conference, 2002, pp. 340–349.

[121] Y. Zorian and V. K. Agarwal, *Optimizing error masking in BIST by output data modification*, J. Electron. Test. **1** (1990), no. 1, 59–71.

[122] Y. Zorian and A. Ivanov, *EEODM: An effective BIST scheme for ROMs*, ITC '90: Proceedings of International Test Conference, Sep 1990, pp. 871–879.

[123] ______, *An effective BIST scheme for ROM's*, IEEE Transactions on Computers **41** (1992), no. 5, 646–653.

[124] Y. Zorian and S. Shoukourian, *Embedded-memory test and repair: infrastructure IP for SoC yield*, IEEE Design and Test of Computers **20** (2003), no. 3, 58–66.