1987

Built-in self test for memory systems /

Steven A. Lerner
Lehigh University

Built-In Self Test for Memory Systems

by

Steven A. Lerner

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science and Electrical Engineering

Lehigh University

1987

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

12/16/86
(date)

[illegible]
Professor in Charge

[illegible]
Chairman of Department

Acknowledgments

I would like to thank Professor A.K. Susskind for his continuous encouragement and valuable comments. I would also like to thank my colleagues and friends for our meaningful discussions about BIST and the memory fault simulator that was developed.

Table of Contents

Table of Contents (Continued)

List of Figures

Abstract

The concept of Built-In Self Test (BIST) is to have the system test itself and provide a "go/nogo" indication. Having a memory system test itself is becoming increasingly important, because memory systems are now embedded within a chip, where the memory system may not be directly controllable or observable at the chip pins. BIST schemes for memory systems are critically analyzed in this study by assessing fault models, memory system functional faults, test patterns/algorithms, and the various related concepts given in the literature. Characteristics of memory test algorithms are given, compared to each other with respect to their fault coverage, and analyzed for their applicability to BIST. The evaluation of different BIST schemes is based on the following: area overhead, fault coverage, test complexity, performance degradation, and the testing of the BIST circuitry. A novel BIST scheme for memory systems is also explored. In addition, for large memory testing and complex algorithms, parallel techniques are discussed. Finally, a guide to help design components of BIST for memory systems is presented.

1. Introduction

Built-In Self Test (BIST) provides a system with the capability of testing itself and gives a "go/nogo" indication. To introduce BIST for memory systems, the following are discussed: architecture and operation of memory systems, overview of testing, and BIST.

1.1 Memory Systems

A memory system consists of the following: a memory address register (MAR), address decoders, an array of memory cells, read/write logic including sense amplifiers, and a memory data register (MDR). See Figure 1.1. The organization of the memory system is N words, each b bits wide. When b is 1, the memory system is considered bit organized; otherwise the memory system is said to be word organized. In a bit organized memory system, each bit can be addressed individually, as compared to a word organized memory system, where all the bits in a particular word have the same address.

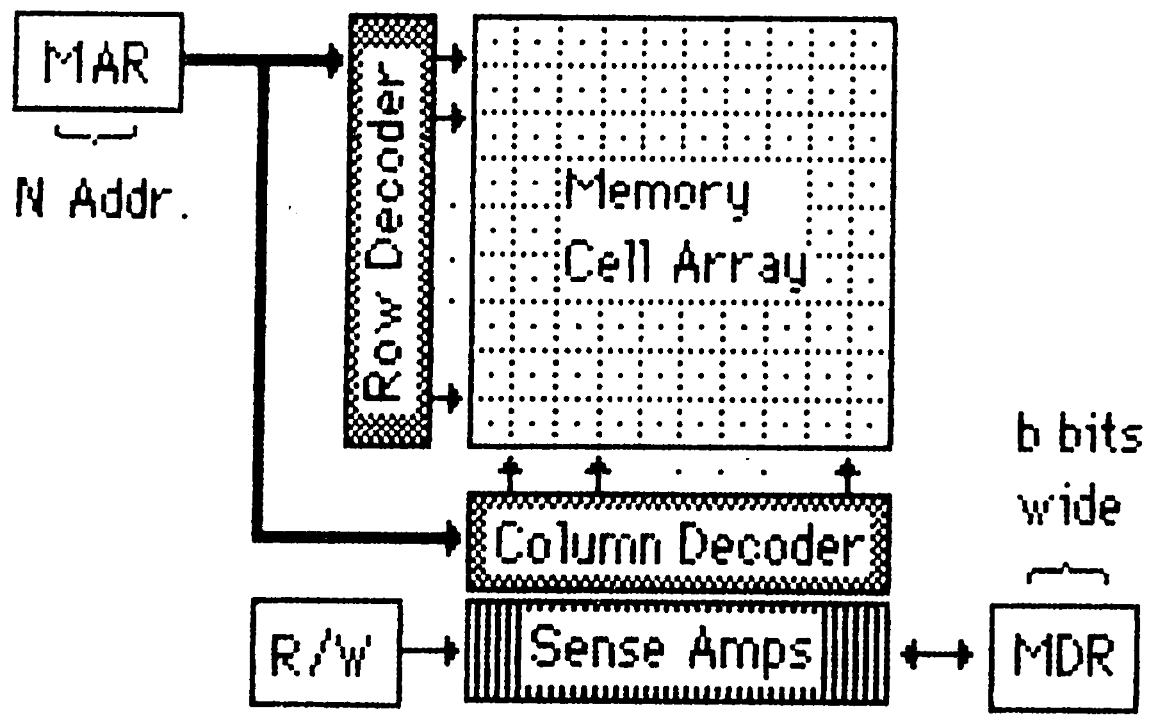


Figure 1.1: Overview of a Memory System: N words by b bits

A memory system operates as follows: a read operation is performed by loading an address into the MAR, setting the read flag, and the resulting word is returned via the MDR; a write operation is accomplished by loading an address into the MAR, loading the MDR with data, and setting the write flag. To understand how the hardware works, the operation of a static CMOS memory system will be reviewed [Mukherjee '86].

A six-transistor static CMOS memory cell and sense amplifier are shown in Figure 1.2. The memory cell is a bistable circuit in which the state of conduction or lack of conduction of a transistor determines the contents of the stored information. Each memory cell contains two pairs of pmos and nmos transistors which are cross coupled. Each transistor pair is such that only one transistor, either the pmos or nmos, is closed at a time. Since this is a bistable circuit, when one side contains a logical 0, by having the pmos open and the nmos closed, the other side would contain a logical 1 by having the pmos closed and the nmos open. For example, if the nmos transistor on the left side is closed and the pmos transistor on the right side is closed, this would cause a 0(1) to be put on the bit line B'(B) when that row is selected. In order to have a large number of cells on a chip, each memory cell is laid out as small as possible. Since the memory cell is small, it can only sink current, not source it, and therefore requires a sense amplifier to sense the information stored and drive the bit lines (B', B).

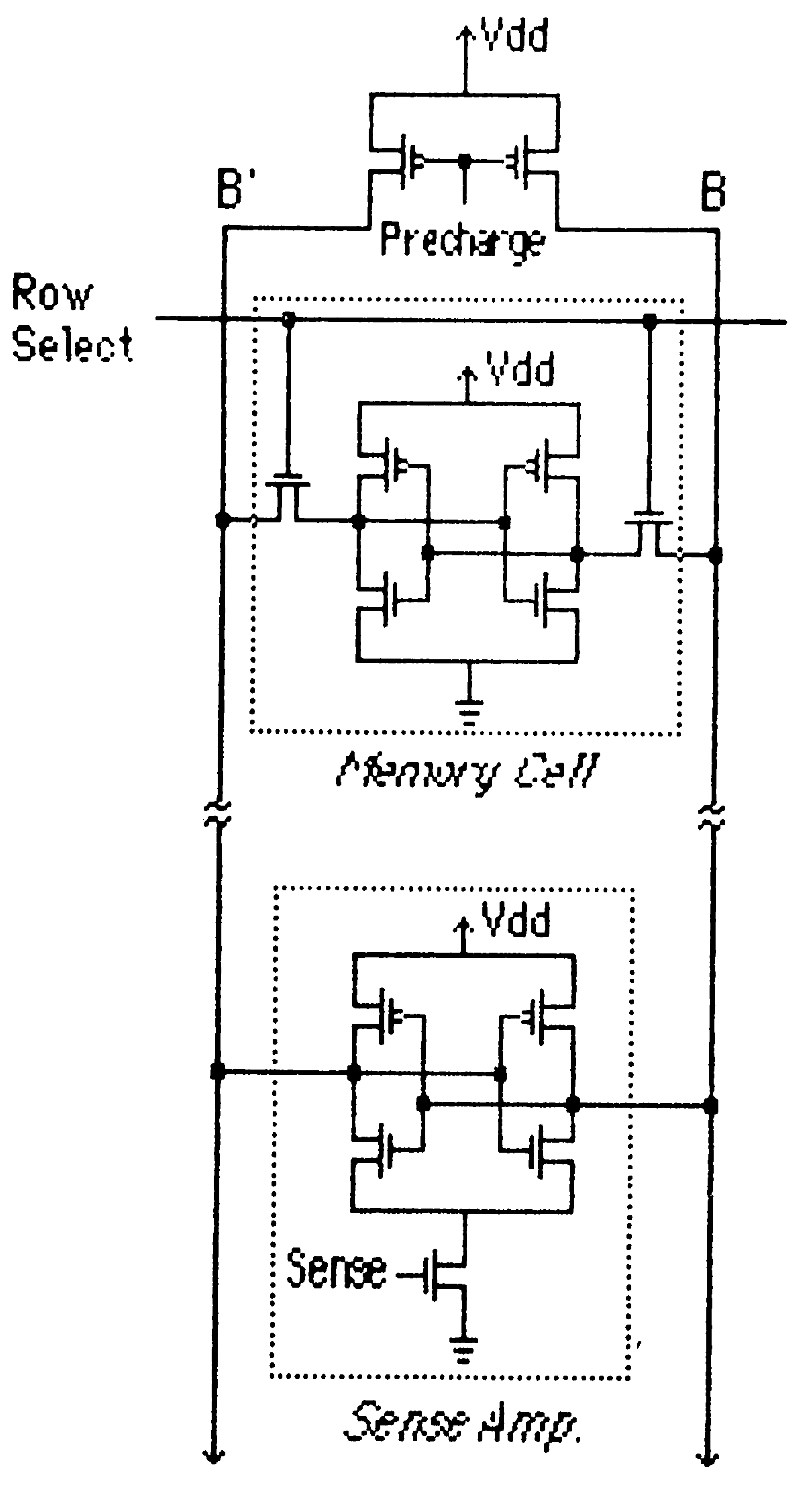


Figure 1.2: Six-transistor Static CMOS Memory Cell and Sense Amp

The read operation of a memory cell is as follows: the sense signal is set low to allow the sense amplifier to float, and simultaneously the precharge signal is set low to precharge the bit lines. (Note, the precharge transistors are made large to allow the bit lines to charge up quickly). After precharging, the precharge signal is set high, thereby making the bit lines behave like capacitors. The row is then selected. Depending on the state of the memory cell, the current in one of the bit lines sinks through the cell and turns on one of the pmos transistors in the sense amplifier (this reinforces the state of the high bit line); the sense line is then set high, which stabilizes the state of the sense amplifier. (The transistors in the sense amplifier are larger than those in the memory cell, and have the capability to drive the bit lines). The data on the bit lines are then selected through the column select, as will be described below.

The write operation is as follows (refer to Figure 1.3). The bit lines (B',B) and the i/o lines (L',L) are precharged by setting the precharge line low; after precharging, the precharge signal is set high which makes the bit lines and the i/o lines behave like capacitors. Next, the R/W' and CE' (chip enable) signals are set low to enable the input transmission gates (T1,T2); thereupon, one of the charged i/o lines discharges via one of the transmission gates (depending on the input data), leaving one line high and the other one low. Next, the column decoder connects one set of bit lines to the i/o lines, which discharges one of the bit lines through the same transmission gate. Then, the sense line of the sense amplifier is set high to help stabilize and drive the bit lines, as discussed previously. Finally, the row decoder selects a particular row, and the data on the bit lines is then written into the memory cell by forcing the bistable circuit to the appropriate state.

For a read operation, the R/W' line is set high and the CE' (chip enable) signal is set low. This disables the input transmission gates (T1,T2) and enables the output transmission gate (T3). For a write operation, the R/W' and CE' signals are set low to enable the input transmission gates and disable the output transmission gate. This control logic, the i/o lines, and the sense amplifiers will be referred to as read/write logic. Note: when reading or writing, all the memory cells in a particular row are selected at once, but only the bit lines of a particular memory cell are connected to the input/output data lines via the selected column decoder line.

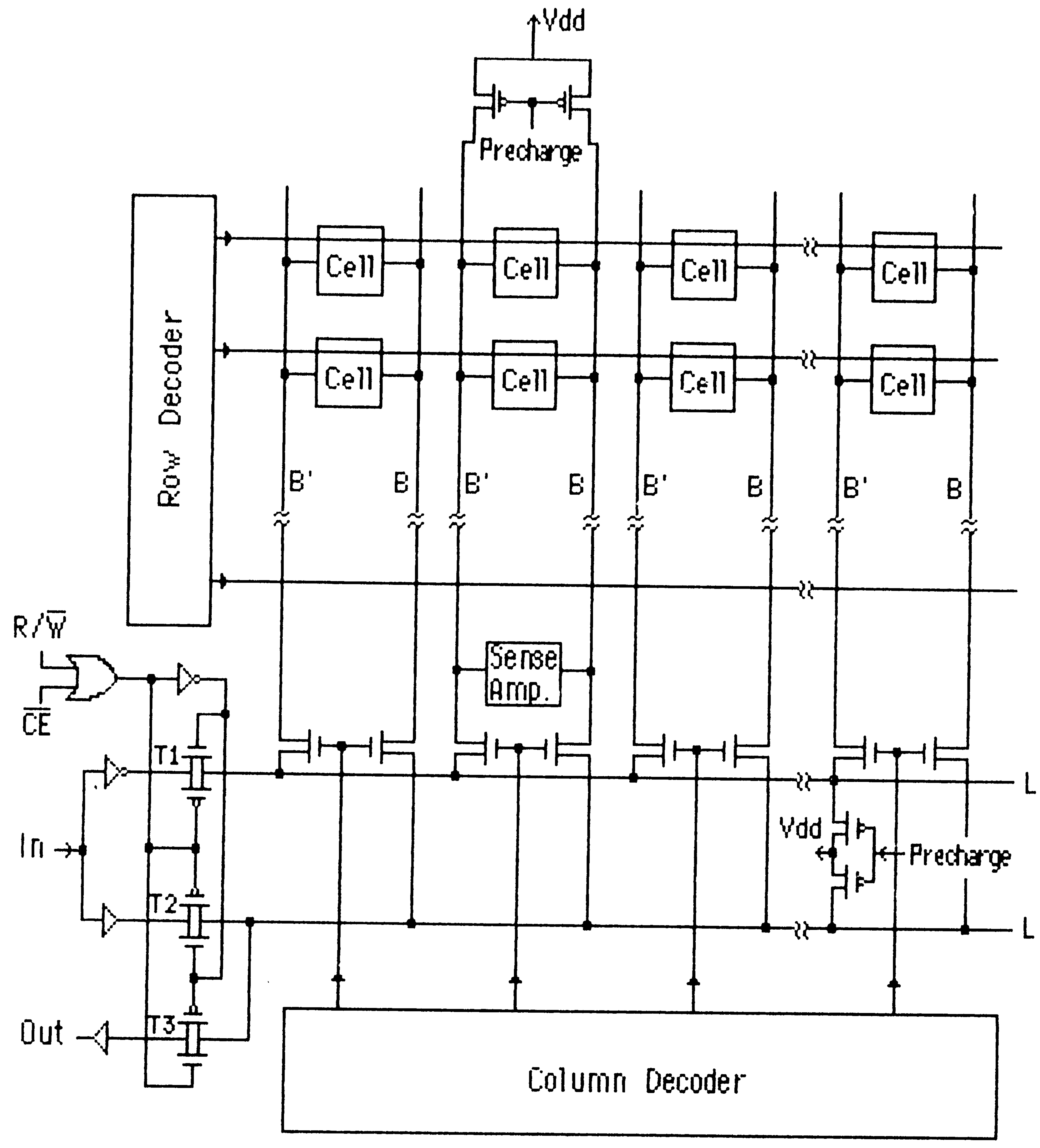


Figure 1.3: Organization of a Static RAM

In almost all applications, word organized memories are used. However, word organized memories at the circuit-board level can be formed by interconnecting bit organized memory chips. See Figure 1.4, where msb stands for most significant bit and cs for chip select.

Functional testing of a memory system, for the most part, can be treated the same for both a board-level memory system and a memory system within a chip. This is because the operation of the memories is the same: select an address and read(write) a word from(into) memory. There are however, some distinctions that can be made. For example, in Figure 1.4, since this system is word organized with individual chips, it is possible to test eight bits in parallel without concern for coupling between bits in the same word. In contrast, if this word organized memory was within a chip, then it makes sense to examine possible coupling within a word, since this type of fault is more likely to occur due to the layout and density of the cell array.

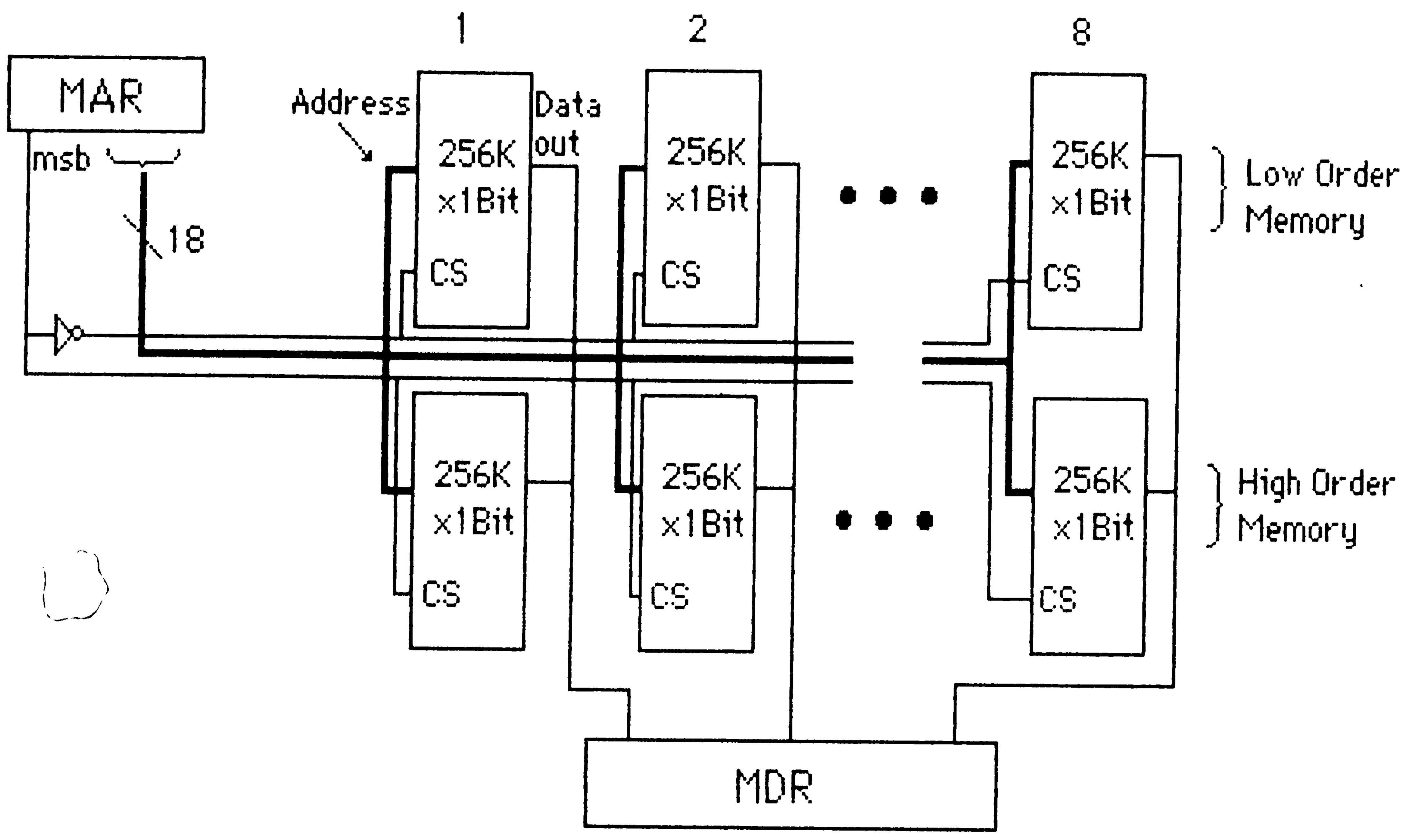


Figure 1.4: 512K x 8 Bit Memory System

In this report, the memory system is assumed to be within a single chip, so that areas of concern such as physical layout of the cell array, decoders, etc., will be addressed. Also, it will be quite easy to apply any of the functional chip-level testing schemes to the board level. Keep in mind that it may be cost effective to implement some built-in functional testing schemes at the board or module level, rather than at the chip level (due to extra pins, area overhead, etc. required otherwise).

1.2 Overview of Testing

Testing circuits consists of three parts: generating test vectors, applying the vectors to the circuit under test (CUT), and comparing the results. A simple block diagram is shown below:

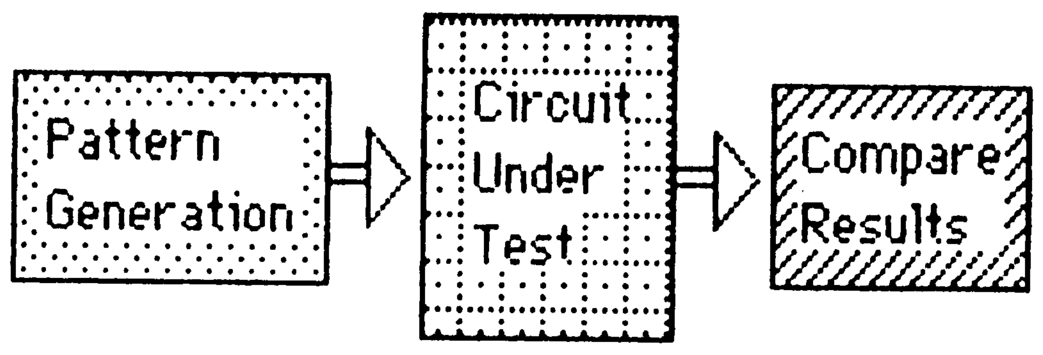


Figure 1.5: Block Diagram of Testing

The set of test vectors is usually generated by expensive automatic test equipment (ATE) and is applied to the CUT. The results of the input test vectors are then compared to a set of correct vectors that is usually stored in the ATE. The generation of the test vectors as well as handling the massive amounts of output data make testing of VLSI chips increasingly complicated, time consuming, and very costly. As the density of memory chips increases, testing them will be a major cost in the overall development. Therefore, new methods need to be developed and implemented. Currently, there is a trend to couple design together with testing, and this has led to Built-In Self Test (BIST).

1.3 Built-In Self Test (BIST)

A system containing BIST can test itself and provide a "go/nogo" indication. This is typically done at the functional level. Most BIST techniques test offline: the system is placed in a self-test mode, and usually a test pattern is generated internally and then either compared to correct results or a signature analysis method is used. In either case, the system gives a go/nogo indication. Another type of BIST is called concurrent testing. This involves the capability of checking the circuit simultaneously with the normal operation of the system. Should a non-correctible error occur, the user would be notified. The major advantage of the concurrent checking method is the ability to detect intermittent faults. For built-in self test of memories, only offline checking will be discussed here.

There are three components of BIST for memory systems: address generation, input data generation, and evaluation circuitry. The evaluation circuitry can be divided into two groups: direct comparison and data compression. Direct comparison is accomplished by comparing expected data to the actual output data. Data compression techniques use either a linear feedback shift register (LFSR) or a multiple input signature register (MISR) to compress the output responses into a signature via polynomial division (refer to [Abraham '85] for more details on signature analysis). See Figure 1.6. LFSR and MISR work similarly: an initial seed value is placed into the register, the testing begins and data compression is performed on the output data. After the test, the resulting signature left in the register is compared to a known signature for the fault-free system.

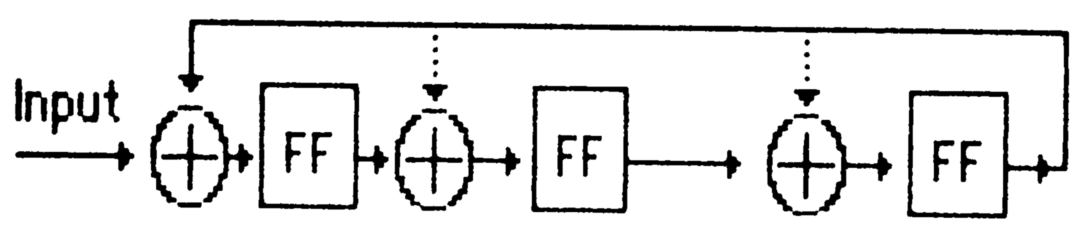


Figure 1.6a: Linear Feedback Shift Register

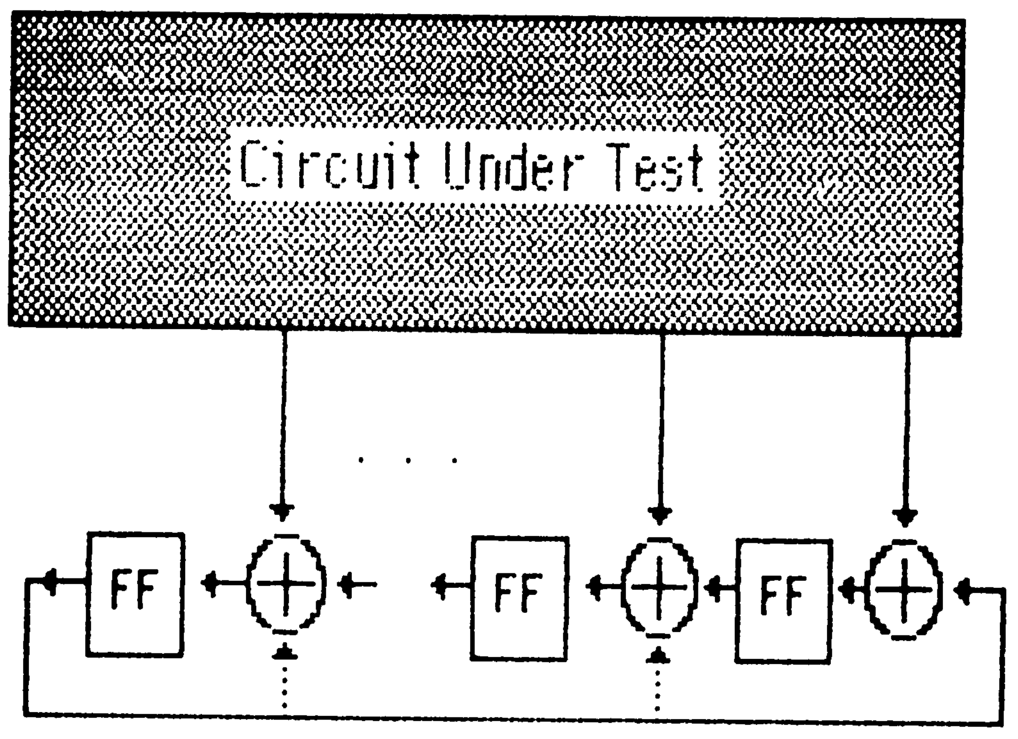


Figure 1.6b: Multiple Input Signature Register

Figure 1.6: Data Compression Techniques

Both methods, direct comparison and data compression, have advantages and disadvantages. Direct comparison gives better fault coverage, because there is no aliasing which is inherent in data compression methods. Also there is no need to initialize the signature register with a seed value, and the signature does not have to be shifted out and compared to results that are stored elsewhere. Data compression techniques, however, have the advantage of testing themselves during the data compression. This is important, since any extra BIST circuitry that is added must also be tested. A memory system with BIST utilizing direct comparison is shown in Figure 1.7a, and Figure 1.7b shows a memory system using data compression.

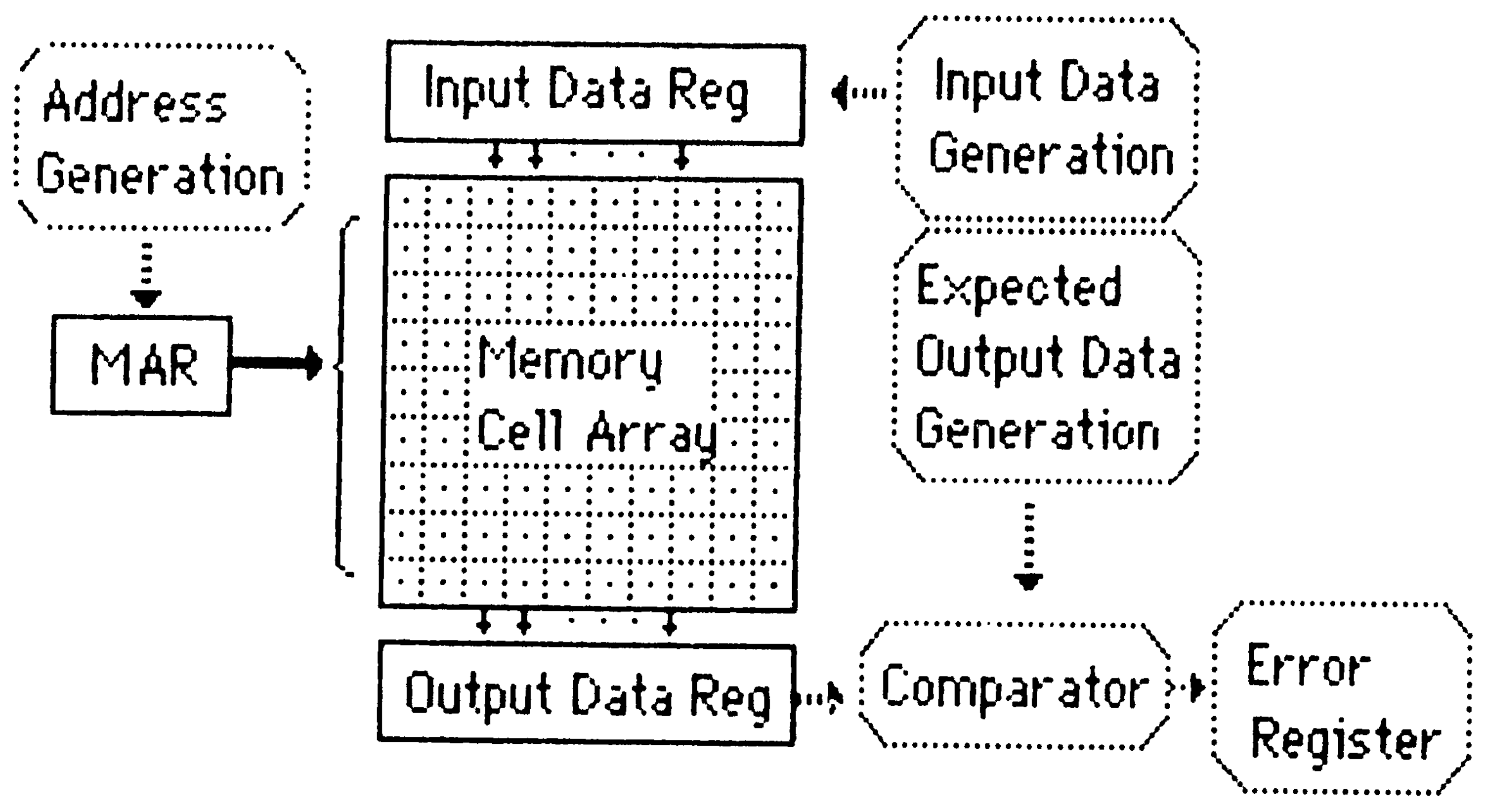


Figure 1.7a: Memory BIST using Direct Comparison

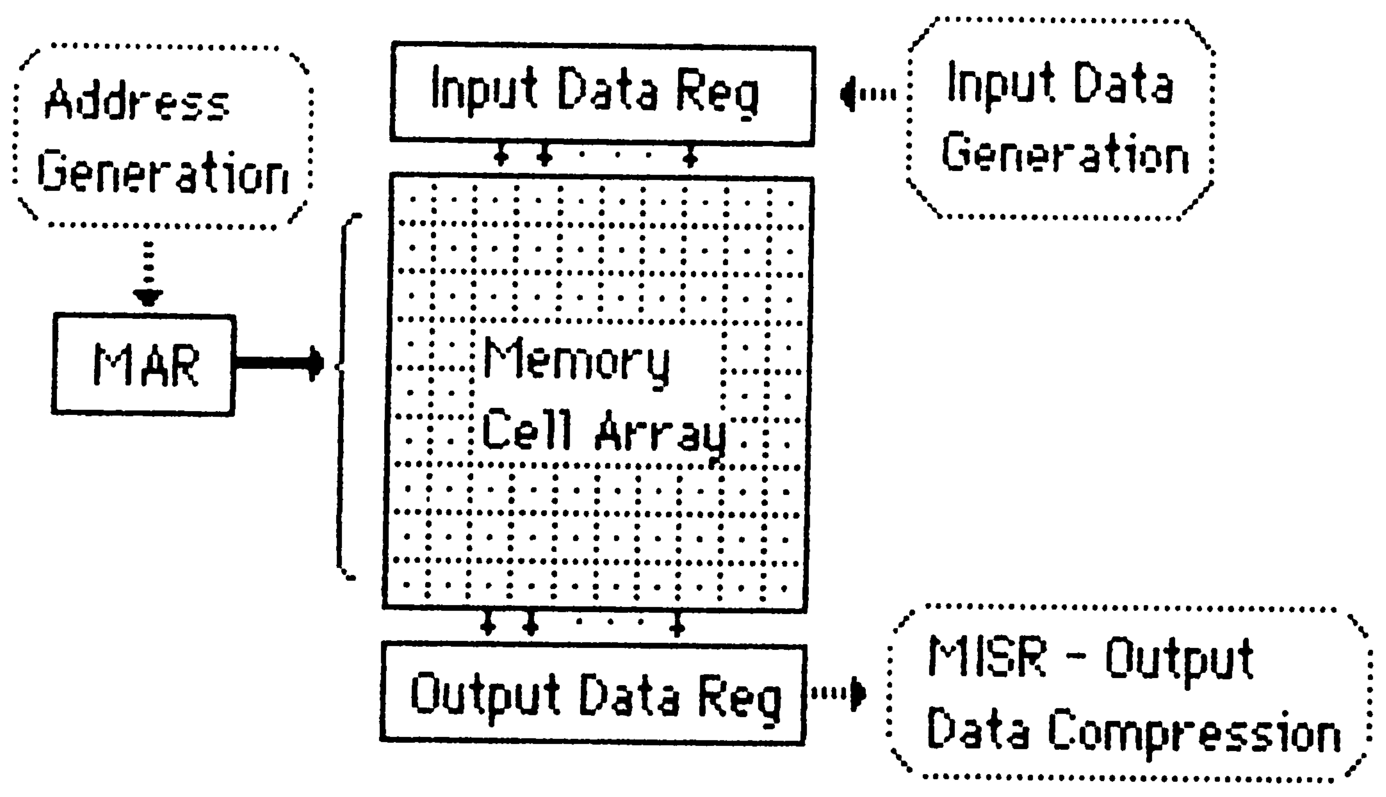


Figure 1.7b: Memory BIST Using Data Compression

Both approaches require some means for address generation and input data generation. The direct comparison method also must be such that the circuitry needed to generate the expected output data is as simple as possible. The approach that should be used will depend on the test algorithm implemented. Ultimately, the assessment of different BIST schemes should be based on the following: area overhead, fault coverage, test time, performance degradation, and testing of the BIST circuitry.

2. Memory Testing

In order to study memory testing, fault models, memory system functional faults, and test patterns/algorithms need to be discussed. In addition, the following terms need to be defined with respect to memory testing. Bit organized memories allow access to each individual cell, which makes it easier to test for cell interaction; word organized memories allow access to a word at a time, which makes testing of coupled cells within a word more difficult since most algorithms only write a word of all 0's or all 1's. Test complexity or test time is given in terms of the number of addressable entities or words in a memory system (for bit organized, the word length is 1). A pseudo-random pattern is a pattern which appears random, but can be reproduced. (In using a LFSR as an address generator, for example, all the addresses of the memory are generated exhaustively, but in a pseudo-random fashion). A memory system, which under some faulty condition reads more than one cell with different values, results in the ANDing (ORing) of the selected cells is considered AND type (OR type). Address Scrambling is when the logical addresses do not map directly to the physical addresses.

2.1 Fault Models

Fault models are logical representations of faults due to physical failures. The common fault models used to test the functionality of a memory system (listed in the order of increasing complexity) are the following: bridging faults, stuck-at faults, hold faults, destructive read faults, transition faults, coupling faults, and pattern-sensitivity faults (PSF).

Bridging Faults Model

There are two types of bridging faults: AND type bridging and OR type bridging. AND type bridging results in dominant 0 while OR type bridging results in dominant 1. For example, if two leads are shorted (a,b), in AND type bridging the value out would be a 0, given that one of the leads was a 0. In OR type bridging, if one of the leads is a 1, then the value out would be a 1. Figure 2.1 shows an AND type bridging fault.

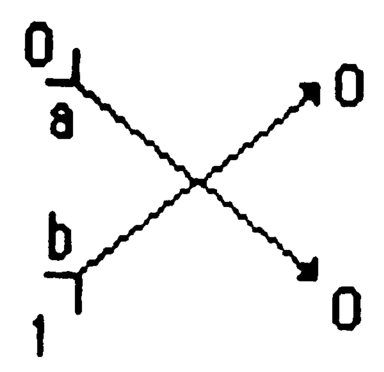


Figure 2.1: AND Type Bridging Fault

Stuck-at Faults Model

The stuck-at fault model assumes that one or more logic values in a memory system (including the address registers, decoders, etc.) cannot be changed. For example, one or more memory cells could be stuck at either a 1 or a 0.

Hold Faults Model *

A memory cell with a hold fault cannot retain either a one or zero state after some amount of time.

Destructive Read Faults Model *

Data in a memory cell is destroyed following a read operation.

Transition Faults Model *

A memory cell fails to undergo a 0->1 or a 1->0 transition.

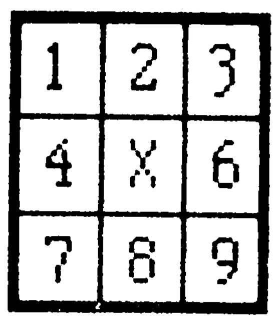
* Distinctive to memory cell array faults

Coupled Faults Model *

A pair of memory cells (i,j) is coupled if a writing a value into one cell of the pair forces the logical state of the other cell. For example, if a 1 is written into cell i which forces cell j to some logical state, say to a 0, then cell i and cell j are said to be coupled. This does not necessarily imply that a similar transition in cell j will influence cell i in a similar manner.

Pattern-Sensitivity Faults (PSF) Model *

The pattern-sensitivity fault model states that a memory cell is dependent on certain patterns of zeros, ones, and/or transitions in the other cells of the memory. This model also includes any failure of a read/write operation involving one cell caused by certain patterns of ones or zeros in the other cells of memory. Note: Coupled faults are a special case of PSF. Normally, for patten-sensitivity faults, one need only be concerned with either the physically adjacent or surrounding cells due to the nature of the physical faults. Figure 2.2 shows a diagram of adjacent and surrounding cells.



Cells 1-9 surround cell X

Cells 2, 4, 6, 8 are adjacent to cell X

Figure 2.2: Adjacent and Surrounding Cells

* Distinctive to memory cell array faults

2.2 Memory System Functional Faults

As was discussed earlier, a memory system consists of the following: a memory address register (MAR), address decoders, array of memory cells, read/write logic, and a memory data register (MDR). Refer back to Figure 1.1. The approach here will be to examine how to test each component of the memory system and to note the equivalence between the component faults and the memory cell array faults. The fault coverage of each algorithm will then be discussed in Test Patterns/Algorithms.

Memory Address Register (MAR)

To test the functionality of the Memory Address Register (MAR), one need only verify that all addresses are accessed. This is because stuck-at or bridging faults in the MAR would cause some address or addresses to be inaccessible. One common approach for verifying that all addresses are accessed is to leave a "trail" of 1's or 0's in each address and then to verify that the trail was left. A trail is made by initializing the memory to some value (say 0), then for all addresses: a) checking that we haven't already been there by reading that value (0), and b) leaving some other value (1) behind as a trail. This method verifies that each address exists and is unique, and it is used in several march test patterns.

Decoders

To test the functionality of the decoders, one must detect stuck-at faults, accessing the wrong cell or cells, and certain coupling faults in the memory cell array. With stuck-at or bridging faults, the decoders will either access no cells (appear as memory cell stuck-at fault), will access the wrong cell or cells (covered in MAR faults), or it will access the correct cell as well as another cell or cells (appear as coupling faults in the memory cell array). In the case of multiple accessing including the correct cell, two or more decoder lines would be activated causing the same value to be written into more than one cell (this is equivalent to coupling of cells with Ai being loaded with some value V and Aj also being loaded with the same value V). In addition, this type of failure also implies that if Ai is coupled to Aj, then Aj is coupled to Ai. Note, when a read operation involves the selection of two cells containing different values, the result will either be a logical 1 or 0 depending if the memory is AND type or OR type.

Read/Write Logic (including sense-amplifiers)

To test the functionality of the read/write logic, one must detect stuck-at faults and certain coupling faults in the memory cell array. Stuck-at faults in the read/write logic will appear as stuck-at faults in the memory cell array. For bit organized memories, bridging faults in the read/write logic will appear as coupling in the memory cell array with the coupled cells being read or written the same value V. Note, the case where a read operation involves the selection of two cells containing different values, the result will either be a logical 1 or 0 depending if the memory is AND type or OR type. For word organized memories, bridging faults can either appear as coupling within a word or coupling between words of the memory array. In addition, this type of failure for either organization implies that if Ai is coupled to Aj, then Aj is coupled to Ai with the same value being written into both cells.

Memory Data Register (MDR)

To test the functionality of the MDR, one must detect stuck-at faults and certain coupling faults in the memory cell array. Stuck-at faults in the MDR will appear as stuck-at faults in the memory cell array, and bridging faults in the MDR will appear as coupling within a word of the memory cell array (for word organized). In addition, this type of failure also implies that if Ai is coupled to Aj, then Aj is coupled to Ai, with the same value being written into both cells.

Memory Cell Array

To test the functionality of the memory cell array, one must detect stuck-at, bridging, transition, hold, destructive read, coupling, and pattern-sensitive faults. Stuck-at faults can be detected by writing a 0 and a 1 into each cell and verifying that a 0 and a 1 have been stored. Bridging faults can be modelled as coupling faults in the following way: Ai is coupled to Aj also implies that Aj is coupled to Ai, and the same value is written into both cells. Transition faults can be detected by forcing each cell to undergo both a 0->1 and a 1->0 transition and verifying each transition. Hold faults can be detected by writing a 0 and a 1 into each cell, waiting some time h, and then verifying that a 0 and a 1 remain. Destructive read faults can be detected by reading each cell twice while each cell contains both a 0 and a 1. Coupling faults can be detected by writing all the combinations of a 0 and a 1 into each pair of cells, and reading after each write operation. Pattern-sensitive faults can be detected by writing patterns into the physical neighborhood (either adjacent or surrounding) of a target cell, and then reading the cells of that neighborhood. Adjacent pattern-sensitive faults will be referred to as APSFs and surrounding pattern-sensitive faults as SPSFs.

As can be seen, to detect all the possible faults within a memory system, one need only cover the memory address register faults and the memory cell array faults. Obviously, to test all the possible memory cell array failures is unrealistic since exhaustively testing a memory requires 2^n operations (where n is the total number of bits in the memory). Therefore, a number of algorithms have been developed to test a memory system based on a fault model or a combination of fault models. Most of the algorithms described in the literature do not cover pattern-sensitive faults. A comparison of these test patterns/algorithms is based on the coverage of the functional faults within a memory system (See Test Patterns/Algorithms).

2.3 Test Patterns/Algorithms

There are many test patterns available for testing RAMs. Each of these patterns offer tradeoffs between test complexity and fault coverage. A memory fault simulator was developed to help analyze the fault coverage of the algorithms. See Appendix A. The following are some of the RAM test patterns (given in the order of increasing complexity): Pseudo-random, MSCAN, Checkerboard, SMTP, Modified Checkerboard, March1, March2, March3, March4, Marinescu, Nair et al., GALPROCO, Walking 1's and 0's, and GALPAT. To be consistent in describing the different algorithms, unless otherwise stated the following convention will be used:

W	Write
R	Read
0	Word of all 0's
1	Word of all 1's
X	Word of mixed 0's and 1's
X'	Complemented X

For describing coupling faults, the following notation wll be used: Cell Ai coupled to cell Aj implies that when cell Ai is loaded with some value V (Ai->V), then cell Aj will be forced to some value X (Aj->X). The asteriks, in the discussion below, mark coupling with the same value (ie; X=V).

Also for consistency, the fault coverage will be in terms of all the possible faults within a memory system (See 2.2 Memory System Functional Faults). Hold faults will not be discussed explicitly since each algorithm can be modified to delay reading some amount of time. The summary of fault coverage will be in terms of the component faults within a bit organized memory system since the coverage of coupling faults for word organized memories is the same for all test algorithms. The summary of fault coverage can easily be extended to word organized memories. A description of each of the algorithms follow:

MSCAN (Memory Scan): O(4n) [Breuer '76] [Abadir '83]

Algorithm:

For all addresses in ascending order: W0, R0, W1, R1

Memory Address Register (MAR) Failure

MAR faults are not detected.

Memory Cell Failure

Stuck-at faults are detected because both a 0 and a 1 are written into all addresses and verified.

Transition faults are not completely detected since each cell fails to undergo a 1->0 transition.

Destructive Read faults are not detected.

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Not Detected	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Not Detected	Detected
*c)	Ai ->0	Aj ->0	Ai < Aj	Not Detected	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected
*e)	Ai ->1	Aj ->1	Ai > Aj	Not Detected	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Not Detected	Detected
*g)	Ai ->0	Aj ->0	Ai > Aj	Not Detected	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Not Detected	Detected

Summary of Fault Coverage

Memory Address Reg	: Not Detected
Decoders	: Stuck-at-0
Read/Write Logic	: Stuck-at
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at

<u>Checkerboard: O(4n)</u> [Sun '84] [Breuer '76]

1. For all addresses; write a checkerboard* pattern (in physical memory)
2. For all addresses; read checkerboard pattern
3. For all addresses; write a complemented checkerboard pattern
4. For all addresses; read complemented checkerboard pattern

* A checkerboard pattern consists of alternating 0's and 1's in odd rows and the complement (alternate 1's and 0's) in the even rows.

Memory Address Register (MAR) Failure

Stuck-at faults in the MAR are not detected by the checkerboard pattern. For example, if the most significant bit of the MAR is stuck-at 0, only the low order addresses are accessed, but these are not detected due to the symmetry of the checkerboard pattern. MAR bridging faults are detected.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 and 1 is written into every other address and verified in step 2. In step 3, a 1 and 0 is written into every other address and verified in step 4.

Transition faults are not necessarily detected since each cell does not undergo both a 0->1 and a 1->0 transition.

Destructive Read faults are not detected.

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word[+++]
*a)	Ai ->1	Aj ->1	Ai < Aj	Not Detected	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Not Detected	Detected in 2,4
*c)	Ai ->0	Aj ->0	Ai < Aj	Not Detected	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected in 2,4
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in 2[+],4[+]	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in 2[+],4[++]	Detected in 2,4
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in 2[+],4[+]	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in 2[+],4[++]	Detected in 2,4

[+] For adjacent cells

[++] For surrounding cells

[+++] Assuming the all 0 and 1 words are written/read to obtain the checkerboard pattern. Depends on physical layout of memory.

Summary of Fault Coverage

Memory Address Reg	: Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Minimal Coupling, Some APSFs, Some SPSFs

SMTP (Simple March Test Pattern): O(5n) [Sun '84]

1. Initialize memory to 0
2. For all addresses in ascending order; R0, W1
3. For all addresses in ascending order; R1, W0

Memory Address Register (MAR) Failure

MAR faults are detected. Step 1 will set all addresses to 0's, and step 2 will verify the MAR by reading the 0's and leaving a trail of 1's.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 is written into all addresses and verified in step 2. In step 2, a 1 is written into all addresses and verified in step 3.

Transition faults are not compeletly detected since although each cell undergoes both a 0->1 and a 1->0 transition (steps 1-3), the 1->0 transition is not verified.

Destructive Read faults are not detected.

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Not Detected	Detected in step 3
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 3	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected in step 2
*e)	Ai ->1	Aj ->1	Ai > Aj	Not Detected	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 3	Detected in step 3
*g)	Ai ->0	Aj ->0	Ai > Aj	Not Detected	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2	Detected in step 2

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Some Transition, Minimal coupling

<u>Modified Checkerboard: O(8n)</u> [Jain '86]

In this algorithm, a checkerboard pattern is written into physical memory such that the bit lines (B', B) of adjacent memory cells contain a checkerboard pattern. The approach is to write the all-1 or all-0 word into the appropriate addresses to achieve this pattern, and therefore requires knowledge of the physcial layout. (The architecture of this memory system is described in section 3.2).

```
1. For all addresses in ascending order
      1A. If address = (1,2,3,4),(9,10,11,12),... Then W0
            Else if address = (5,6,7,8),(13,14,15,16),... Then W1
      1B. Read address (either R0 or R1 depending on address)
2. For all addresses in ascending order
      2A. Read address
      2B. If address = (1,2,3,4),(9,10,11,12),... Then W1
            Else if address = (5,6,7,8),(13,14,15,16),... Then W0
3. For all addresses in ascending order
      3A. Read address
      3B. Read addresses descending order
4. For all addresses in descending order
      4A. If address = (1,2,3,4),(9,10,11,12),... Then W0
            Else if address = (5,6,7,8),(13,14,15,16),... Then W1
      4B. Read address
```

Memory Address Register (MAR) Failure

Stuck-at faults in the MAR are not detected due to the symmetry of the checkerboard pattern. MAR bridging faults are detected.

Memory Cell Failure

Stuck-at faults are detected since both a 0 and a 1 are written into all addresses and verified.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-4), each followed by a read operation.

Destructive Read faults are detected since each cell is read twice after containing both a 0 and a 1.

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word+++
*a)	Ai ->1	Aj ->1	Ai < Aj	Not Detected	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Not Detected	Detected in 1,2
*c)	Ai ->0	Aj ->0	Ai < Aj	Not Detected	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected in 1,2
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected 2-4+	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected 2-4++	Detected in 1,2
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected 2-4+	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected 2-4++	Detected in 1,2

+ For adjacent and surrounding cells but in different rows

++ For cells within the same row

+++ Assuming the all 0 and 1 words are written/read to obtain the checkerboard pattern. Depends on physical layout of memory.

Summary of Fault Coverage

Memory Address Reg	: Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, min Coupling, Destructive Read, Transition, Some APSFs, Some SPSFs

<u>March 1: O(9n)</u> [Green '86]

1. Initialize memory to 0
2. For all addresses in ascending order; R0, W1
3. For all addresses in ascending order; R1, W0
4. For all addresses in descending order; R0, W1
5. For all addresses in descending order; R1, W0

Memory Address Register (MAR) Failure

Steps 1-3 are identical to the SMTP; therefore March 1 will detect any MAR failures.

Memory Cell Failure

Stuck-at faults are detected since steps 1-3 are identical to SMTP.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-4), each followed by a read operation.

Destructive Read faults are not detected.

Coupled Cells -

				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Detected in step 5	Detected in step 3
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 3	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected in step 2
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 4	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 3	Detected in step 3
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 5	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2	Detected in step 2

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Transition, most Coupling

<u>March 2: O(11n)</u> [Green '86]

1. Initialize memory to 0
2. For all addresses in ascending order; R0
3. For all addresses in ascending order; R0, W1
4. For all addresses in ascending order; R1
5. For all addresses in ascending order; R1, W0
6. For all addresses in descending order; R0, W1
7. For all addresses in descending order; R1, W0

Memory Address Register (MAR) Failure

MAR failures are detected because step 3 will leave a trail of 1's, and step 5 will verify the trail of 1's.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 is written into all addresses and verified in step 2. In step 3, a 1 is written into all addresses and verified in step 4.

Transition faults are detected since each cell undergoes both a 0->1 and 1->0 transition (steps 1,3,5,6), each followed by a read operation.

Destructive Read faults are detected since each cell is read twice while containing both a 0 and a 1 (steps 2,3,4,5).

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 3	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Detected in step 7	Detected in step 4
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 5	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected in step 2
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 6	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 4	Detected in step 4
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 7	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2	Detected in step 2

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Destructive Read, Transition, Most Coupling

<u>March 3: O(14n)</u> [Breuer '76]

1. Initialize memory to 0
2. For all addresses in ascending order; R0, W1, R1
3. For all addresses in descending order; R1, W0, R0
4. Initialize memory to 1
5. For all addresses in ascending order; R1, W0, R0
6. For all addresses in descending order; R0, W1, R1

Memory Address Register (MAR) Failure

MAR faults are detected. Step 1 will set all addresses to 0. Step 2 will verify the MAR by reading the 0's and leaving a trail of 1's.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 is written into all addresses and verified in step 2. In step 2, a 1 is written into all addresses and then verified.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-3), each followed by a read operation.

Destructive Read faults are detected since each cell is read twice while containing both a 0 and a 1 (steps 2,3,5,6).

Coupled Cells -

				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Not Detected	Detected in step 2
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 5	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Not Detected	Detected in step 2
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 6	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 3	Detected in step 2
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 3	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2	Detected in step 2

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Destructive Read, Transition, Some Coupling

<u>March 4: O(15n)</u> [Pradham '86]

1. Initialize memory to 0
2. For all addresses in ascending order; R0, W1, W0, W1
3. For all addresses in ascending order; R1, W0, W1
4. For all addresses in descending order; R1, W0, W1,W0
5. For all addresses in descending order; R0, W1, W0

Memory Address Register (MAR) Failure

MAR faults are detected. Step 2 will leave a trail of 1's and step 3 will verify that the trail of 1's was left.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 is written into all addresses and verified in step 2. In step 2, a 1 is written into all addresses and verified in step 3.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-5), each followed by a read operation.

Destructive Read faults are not detected.

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Detected in step 3	Detected in step 3
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 3	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Detected in step 2	Detected in step 2
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 5	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 3	Detected in step 3
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 3	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2	Detected in step 2

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Transition, Coupling

Marinescu: O(17n) [Marinescu '82] [Nicolaidis '85]

1. Initialize memory to 0
2. For all addresses in ascending order; R0, W1, W0, W1
3. For all addresses in ascending order; R1, W0, R0, W1, R1
4. For all addresses in descending order; R1, W0, W1, W0
5. For all addresses in descending order; R0, W1, W0

Memory Address Register (MAR) Failure

MAR faults are detected. Step 2 will leave a trail of 1's and step 3 will verify that the trail of 1's was left.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 is written into all addresses and verified in step 2. In step 2, a 1 is written into all addresses and verified in step 3.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-3), each followed by a read operation.

Destructive Read faults are not completely detected: each cell is read twice after containing a 1, but not after containing a 0.

Coupled Cells -

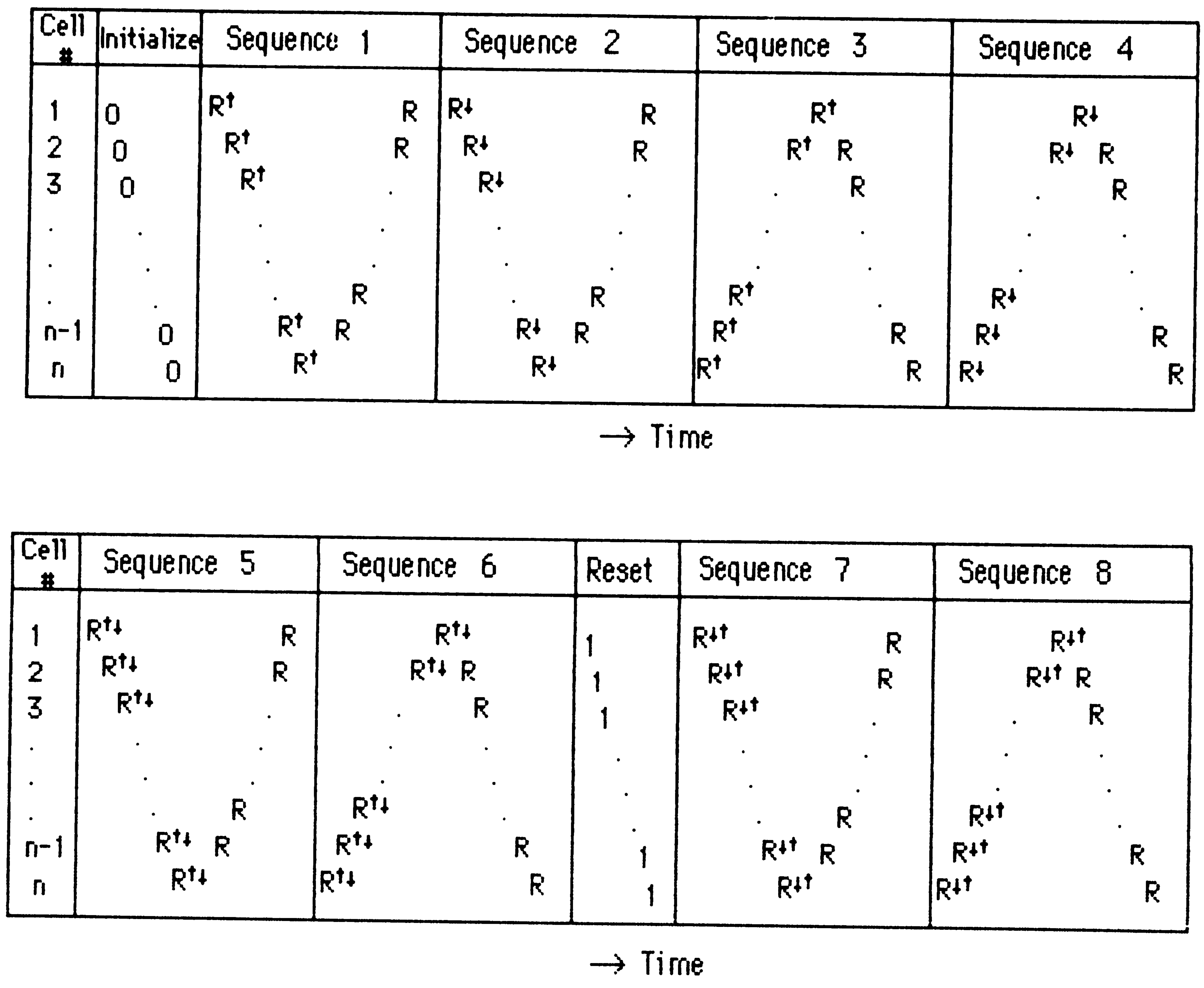
				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Detected in step 3	Detected in step 3
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 3	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Detected in step 2	Detected in step 2
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 5	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 3	Detected in step 3
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 3	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2	Detected in step 2

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Some Destructive Read, Transition, Coupling

Nair et al: O(30n) [Nair '78]

The algorithm is shown in the figure below:



Key
R Read
↑ Write a 1
↓ Write a 0

Nair, Thatte, and Abraham's Testing Procedure [Nair et al. 1978]

Figure 2.3: Nair, Thatte, and Abraham's Algorithm

Memory Address Register (MAR) Failure

MAR faults are detected in sequence 1 by leaving a trail 1's.

Memory Cell Failure

Stuck-at faults are detected. Initially, a 0 is written into all addresses, and then verified in sequence 1. Sequence 1 also writes a 1 into all addresses and verifies it.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (initalize through sequence 2), each followed by a read operation.

Destructive Read faults are detected since each cell is read twice while containing both a 0 and a 1 (sequences 1-3).

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in Sequ. 1	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Detected in Sequ. 3	Detected Sequ. 1
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in Sequ. 2	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Detected in Sequ. 4	Detected Sequ. 1
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in Sequ. 3	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in Sequ. 1	Detected Sequ. 1
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in Sequ. 4	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in Sequ. 1	Detected Sequ. 1

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at

Memory Cell Array : Stuck-at, Bridging, Destructive Read, Transition, Coupling

GALPROCO: $O(4*(3n+2*n^{3/2}))$ [Daehn '86]

```
1. Initialize memory to 0
2. For all columns:
        2. For all rows:
            2a. R0
            2b. W1
            2c. R0 for all other columns, R1 for Test bit
            2d. R0 for all other rows, R1 for Test bit
            2e. W0
            2f. R0
3. For all addresses (rows & columns) R0
4. Initialize memory to 1
5. For all columns:
        5. For all rows:
            5a. R1
            5b. W0
            5c. R1 for all other columns, R0 for Test bit
            5d. R1 for all other rows, R0 for Test bit
            5e. W1
            5f. R1
6. For all addresses (rows & columns) R1
```

Memory Address Register (MAR) Failure

MAR faults are detected.

Memory Cell Failure

Stuck-at faults are detected. In step 1, a 0 is written into all addresses and verified in step 2a. In step 2b, a 1 is written into all addresses and verified in step 2c.

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-2f), each followed by a read operation.

Destructive Read faults are detected since each cell is read twice while containing both a 0 and a 1 (steps 2c$^+$, 2d$^+$, 3, 5c$^+$, 5d$^+$, 6).

Coupled Cells -				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in 2a, 2c$^+$,2d$^+$	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Detected in 5a,5c$^+$,5d$^+$	Detected in 2C
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in 5a,5c$^+$,5d$^+$	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Detected in 2a, 2c$^+$,2d$^+$	Detected in 2F
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in 3, 2c$^+$,2d$^+$	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in 5a,5c$^+$,5d$^+$	Detected in 2C
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in 6,5c$^+$,5d$^+$	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in 2a, 2c$^+$,2d$^+$	Detected in 2F

$^+$ For coupling within the same row or column.

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Destructive Read, Transition, Coupling, Some APSFs

<u>Walking 1's and 0's:</u> $O(2n^2 + 6n)$ [Breuer '76]

1. Initialize memory to 0
2. For all addresses in ascending order:
   - 2a. W1
   - 2b. R0 for all other addresses (Tests that no cell is disturbed)
   - 2c. R1 (Tests that the test bit is still correct)
   - 2d. W0
3. Initialize memory to 1
4. For all addresses in ascending order:
   - 4a. W0
   - 4b. R1 for all other addresses (Tests that no cell is disturbed)
   - 4c. R0 (Tests that the test bit is still correct)
   - 4d. W1

Memory Address Register (MAR) Failure

MAR failures are detected.

Memory Cell Failure

Stuck-at faults are detected since both a 0 and a 1 are written into all addresses and then verified (steps 1-2d).

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-2d), each followed by a read operation.

Destructive Read faults are detected since each cell is read twice while containing both a 0 and a 1 (steps 2b, 4b).

Coupled Cells -

				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2b	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Mostly Detected 4b	Detected in 2c
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 4b	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Mostly Detected 4b	Detected in 2b
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 2b	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 4b	Detected in 2c
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 4b	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2b	Detected in 2b

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Destructive Read, Transition, Most Coupling, Some APSFs, Some SPSFs

__GALPAT:__ $O(2n^2 + 8n)$ [Breuer '76] [Pradham '86]

1. Initialize memory to 0
2. For all addresses in ascending order:
    - 2a. W1
    - 2b. For all other addresses; R0 and verify R1 for test address
    - 2c. R1 (Tests that the test bit is still correct)
    - 2d. W0
3. Initialize memory to 1
4. For all addresses:
    - 4a. W0
    - 4b. For all other addresses; R1 and verify R0 for test address
    - 4c. R0 (Tests that the test bit is still correct)
    - 4d. W1

Memory Address Register (MAR) Failure

MAR failures are detected.

Memory Cell Failure

Stuck-at faults are detected. Both a 0 and a 1 is written into all addresses and verified (steps 1-2d).

Transition faults are detected since each cell undergoes both a 0->1 and a 1->0 transition (steps 1-2d).

Destructive Read faults are detected since each cell is read twice while containing both a 0 and a 1 (steps 2b, 2c, 4b, 4c).

Coupled Cells -

				Ai and Aj in different words	Ai and Aj within a word
*a)	Ai ->1	Aj ->1	Ai < Aj	Detected in step 2b	Not Detected
b)	Ai ->1	Aj ->0	Ai < Aj	Mostly detected 4b	Detected in 2b
*c)	Ai ->0	Aj ->0	Ai < Aj	Detected in step 4b	Not Detected
d)	Ai ->0	Aj ->1	Ai < Aj	Mostly detected 2b	Detected in 2b
*e)	Ai ->1	Aj ->1	Ai > Aj	Detected in step 2b	Not Detected
f)	Ai ->1	Aj ->0	Ai > Aj	Detected in step 4b	Detected in 2b
*g)	Ai ->0	Aj ->0	Ai > Aj	Detected in step 4b	Not Detected
h)	Ai ->0	Aj ->1	Ai > Aj	Detected in step 2b	Detected in 2b

Summary of Fault Coverage

Memory Address Reg	: Stuck-at & Bridging
Decoders	: Stuck-at & Bridging
Read/Write Logic	: Stuck-at & Bridging
Memory Data Reg	: Stuck-at
Memory Cell Array	: Stuck-at, Bridging, Destructive Read, Transition, Most Coupling, Some APSFs, Some SPSFs

2.4 Comparison of Algorithms

The chart shown in Figure 2.4 compares the algorithms previously discussed in Test Patterns/Algorithms. This chart summarizes the following for each algorithm: the test complexity in terms of the number of words in memory, the memory cell array fault coverage, an estimate of the area overhead required for BIST circuitry, the estimated time to test a 256K RAM with 200ns access time, and miscellaneous comments. Unless otherwise stated, MAR faults are covered by the algorithms. As can be seen, there are tradeoffs between the fault coverage, the test complexity, and the overhead in terms of BIST circuitry. Typically, to obtain better fault coverage, the tradeoff is increased test complexity and area overhead. The estimated time column shows which tests run in a reasonable amount of time. To reduce the test time for complex algorithms, such as GALPAT, parallel techniques can be implemented (See Speed-up Techniques: Parallelism). Note: for other memory sizes, for example 8K, the test time would simply be reduced by a factor of 32. Analyzing the tradeoffs between algorithms, the following observations are made.

In order to cover stuck-at, bridging, transition, destructive read, and coupling faults only O(n) algorithms are required. For pattern sensitive faults (PSFs), $O(n^{3/2})$ or $O(n^2)$ algorithms are recommended when the mapping of logical to physical addresses is not available or practical. The test times with these algorithms are too long for large memories, and therefore need to be implemented using parallel techniques. The problems associated with algorithms that require knowledge of the physical layout are due to address scrambling and row/column replacement. To handle address scrambling, extra circuitry is needed to map the logical addresses to the physical addresses. During the manufacturing process, some faulty rows and/or columns maybe logically replaced by spare rows or columns. This makes testing the interaction of cells within the physical layout nearly impossible. When knowledge of the logical to physical addresses is available, and the test time requirement is low, then a modified checkerboard approach is recommended.

Test Patterns	Complexity	Fault Coverage*	Overhead**	Est. Time***	Comments
Pseudo-Random	4n	A, $\underline{B}$, $\underline{E}$, $\underline{F}$, $\underline{G}$	Small	.21S	1,2
MSCAN	4n	A	Small	.21S	2
Checkerboard	4n	A, B, $\underline{C}$, $\underline{E}$, $\underline{F}$	Medium	.21S	2,3
SMTP	5n	A, B, $\underline{C}$, $\underline{E}$	Small	.26S	
Modified Checkerboard	8n	A, B, C, D, $\underline{E}$, $\underline{F}$, $\underline{G}$	Medium	.42S	2,3
March1	9n	A, B, C, $\underline{E}$	Small	.47S	
March2	11n	A, B, C, D, $\underline{E}$	Small	.58S	
March3	14n	A, B, C, D, $\underline{E}$	Small	.73S	
March4	15n	A, B, C, E	Small	.79S	
Marinescu	17n	A, B, C, $\underline{D}$, E	Small	.89S	
Nair et al.	30n	A, B, C, D, E	Medium	1.57S	
GALPROCO	$12n+8n^{3/2}$	A, B, C, D, E, $\underline{F}$	Large	3.59M	4
Walking 1's/0's	$2n^2 + 6n$	A, B, C, D, $\underline{E}$, $\underline{F}$, $\underline{G}$	Large	7.64H	5
GALPAT	$2n^2 + 8n$	A, B, C, D, $\underline{E}$, $\underline{F}$, $\underline{G}$	Large	7.64H	5

A-Stuck-at B-Bridging C-Transition D-Destructive Read E-Coupling F-APSF G-SPSF

_ Not completely detected

* Memory Cell Array Fault Coverage (bit organized)

** Estimated overhead of BIST circuitry

*** Based on 256K with 200ns access time

Comments

1 Don't know how many patterns are necessary to obtain indicated coverage. Coverage is estimated.

2 MAR faults are not completely detected.

3 Knowledge of how physical positions map to their logical addresses is required.

4 Must have capability to specify row and column addresses separately.

5 Test complexity too large for BIST of large memories without using parallel techniques.

Figure 2.4: Comparison Chart of Algorithms

3. Built-In Self Testing Memories

To understand built-in self test for memories, memory fault models along with memory test patterns/algorithms needed to be discussed. To evaluate the current state of built-in self test for memories, it will be necessary to study and analyze the approaches currently being suggested in the literature [Bardell '85] [Daehn '86] [Jain '86] [Kinoshita '86] [Nicolaidis '85] [Sun '84] [Westcott '81] [You '84]. The following approaches with their advantages and disadvantages are discussed: Daehn & Gross [Daehn '86], Jain & Stroud [Jain '86], and Sun & Wang [Sun '84].

3.1 Daehn, Wilgried & Gross, Josef: A Test Generator IC for Testing Large CMOS-RAMs

Overview

A test generator IC for testing large CMOS-RAMs utilizing the GALPROCO $O(n^{1.5})$ algorithm is discussed. The paper analyzes the effects of physical defects on RAM behavior, discusses existing test algorithms, proposes an algorithm which is independent of the physical layout of the RAM, and presents the implementation in silicon. The implemented test concept is applicable to both board and chip level testing.

Approach

The RAM is assumed to have the organization of Figure 1.1, except that there are two address registers: a separate row and column address register. The physical defects within all the functional blocks of the RAM transform themselves to logical fault models in the following way:

Physical Defects	Logical Fault Model
opens	CMOS stuck-open
short to GND	stuck-at-0
short to VDD	stuck-at-1
bridges	asymetrically wired AND, OR forced level

These logical fault models are then transformed into the following funtional level model for a RAM:

Cells can be s-a-0, s-a-1

Simple decoding error: a given address x selects a cell≠x

Sequential decoding error: an address x selects a cell x, a following arbitrary address y selects a cell≠y (effects of a CMOS stuck open fault ir. the decoder)

Pattern sensitivity in a column or row

Pattern sensitivity in a physical neighborhood (not in a column or row) (Not covered in this approach)

Pattern sensitivity between any other cell pair (Not covered in this approach)

A test procedure was developed to meet the above requirements plus the following: independence of row/column arrangement of the cell array, refresh of dynamic RAMs during test execution without additional cycles, and acceptable execution time. Below is a block diagram of the test processor IC.

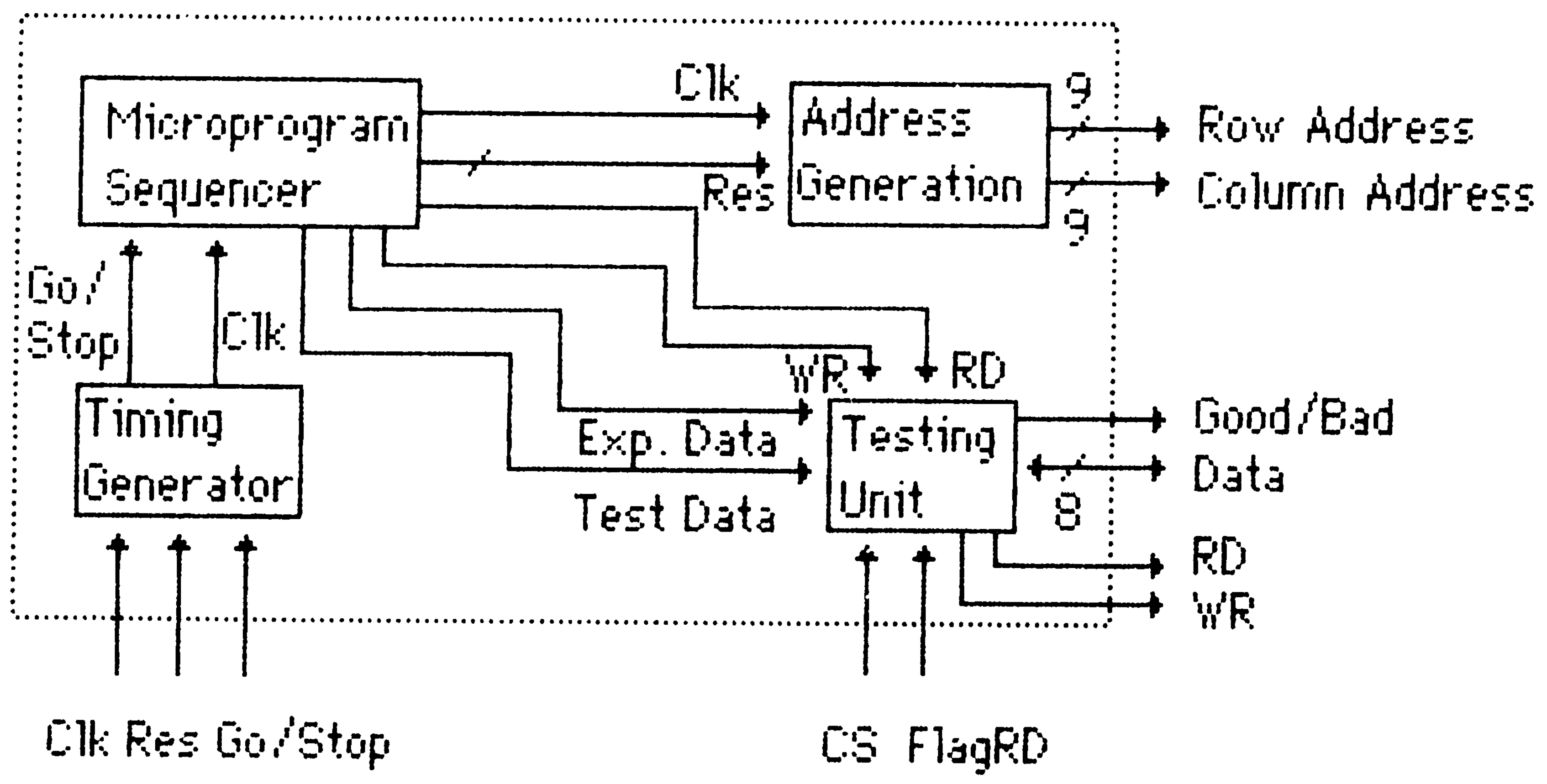


Figure 3.1: Block Diagram of the Test Processor IC

The timing generator organizes the timing and controls the start/stop logic of the chip. The execution of the test procedure is controlled by a finite state machine, which acts as a microprogram sequencer. The address generating unit consists of four linear feedback (9-bit) shift registers (LFSR) because they require less silicon area than conventional counting circuits. These are designed such that they generate maximum length sequences for both row and column addresses including address 0. The testing unit compares incoming data from the RAM under test with expected data, generated by the control unit. This is done for eight bits in parallel.

Advantages

At the board level there is little overhead. This approach is also independent of the physical layout while providing good coverage.

Disadvantages

Requires separate row and column address registers. The test time in the future maybe a problem since the test complexity is $O(n^{3/2})$. Also, this approach covers a limited amount of PSFs. Estimate high overhead for BIST within a chip.

Comments

This approach is designed for a separate test processor chip. The GALPROCO algorithm covers more faults than stated in the proposed fault model. For example, destructive read faults. See Test Patterns/Algorithms and Comparison Chart for a description of the coverage.

3.2 Jain, Sunil & Stroud, Charles: Built-in Self Testing of Embedded Memories

Overview

Two test schemes, using the pseudo-random and checkerboard algorithm, are presented as a built-in self test (BIST) method for testing embedded memories. The fault model used in both test schemes can be summarized as follows: memory cell array faults (stuck-at, transition, hold, destructive read, adjacent bridging, adjacent coupling), decoder stuck-at faults, and adjacent data input/output line coupling. The adjacent faults largely depend on the physical layout of the memory cell array. The RAM used in this illustrative example is 32 words, accessed with 5 address bits, each word containing four data bits. The paper also discusses the test pattern requirements, data compression techniques, implementation, and an evaluation of both test schemes.

Approach

Approach1: Pseudo-random algorithm

In this scheme, the input stimulus to supply the address to the RAM is also used as the input data, such that each memory word contains a unique data word (pseudo-random). The algorithm is shown below:

1. For all addresses in ascending order, Wx (where x=address)
2. For all addresses in ascending order, Rx (where x=address)
3. For all addresses in ascending order, Wx'
4. For all addresses in ascending order, Rx'
5. For all addresses in descending order, Rx'
6. For all addresses in ascending order, Wx
7. For all addresses in ascending order, Rx
8. For all addresses in descending order, Rx

Implementation of this scheme consists of a binary counter to supply the address, input data, and three control signals to direct the test sequence. The output data is compressed via a multiple input signature register (MISR). The signature register is enabled(disabled) during a read(write) operation and is controlled by the same signal that controls the read/write to the RAM. When the self-testing is initiated, the binary counter is initialized to zero and the MISR is initialized to a seed value. A MISR is used to compress the output because with a comparison method all the expected data inputs would have to be routed to the output for comparison, which would increase area overhead. Below is an overview of this scheme:

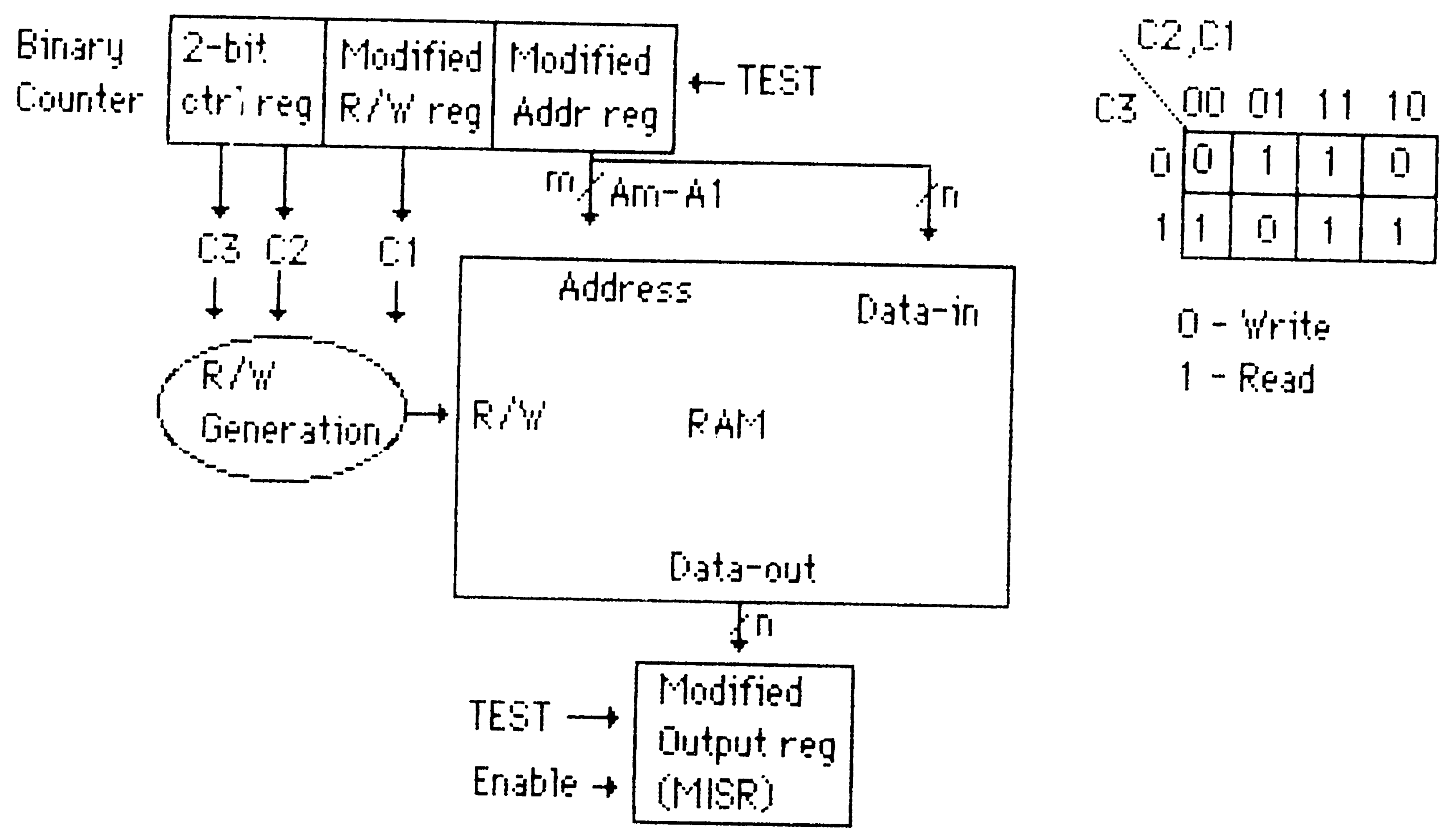


Figure 3.2: Test Scheme 1

Approach 2: Checkerboard algorithm

In this scheme, a checkerboard pattern is written into physical memory such that the bit lines (B', B) of adjacent memory cells contain a checkerboard pattern. The approach is to write the all-1 or the all-0 word into the appropriate addresses to achieve this pattern. The algorithm is discussed in section 2.3 Test Patterns/Algorithms (modified checkerboard algorithm) using the architecture of the memory system described previously.

Implementation of this scheme is very similar to the one previously discussed except that the binary counter is modified to generate the checkerboard pattern. The output response can be either compressed or compared to expected data. See below:

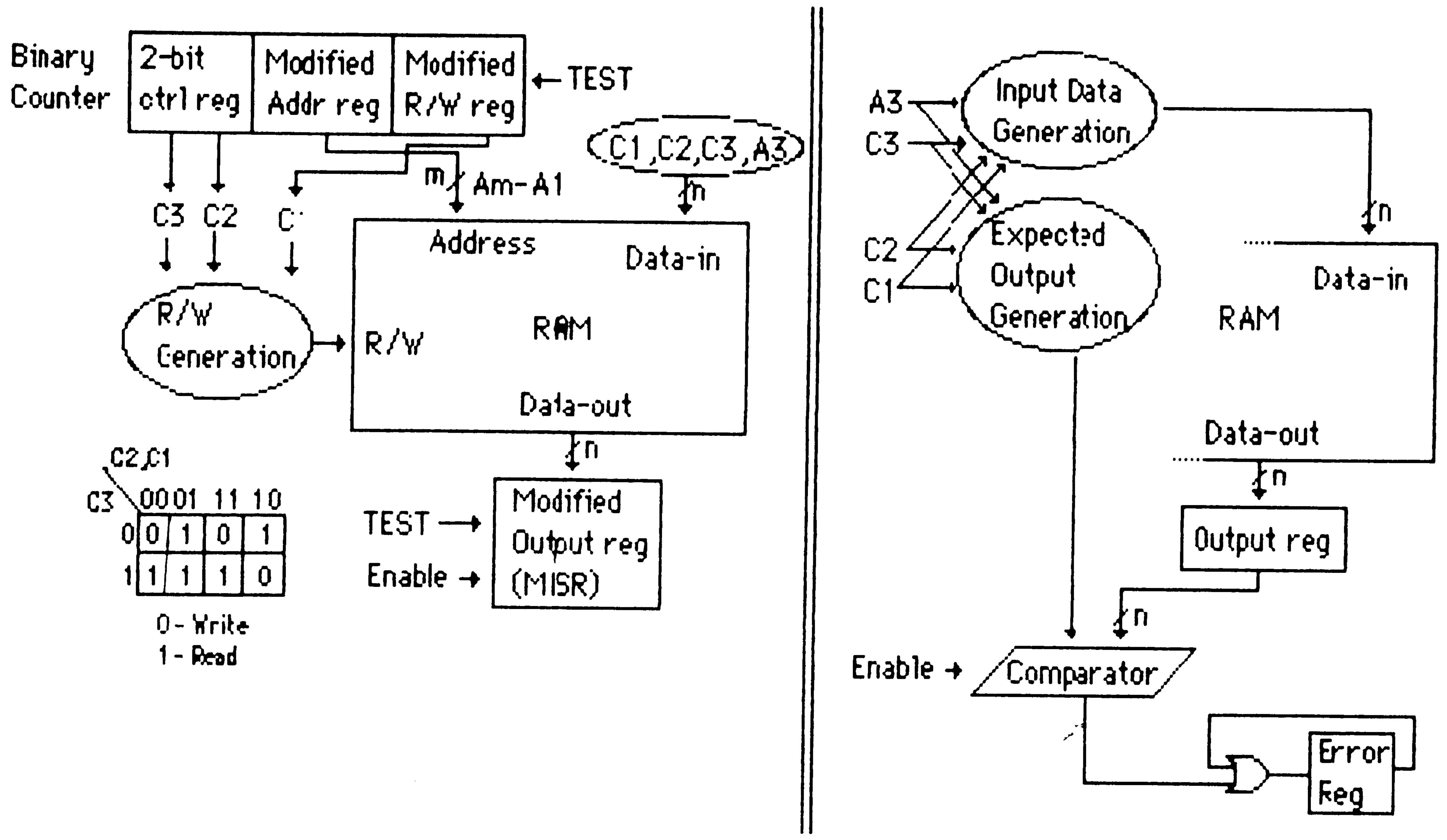


Figure 3.3: Scheme 2 with a) Data Compression and b) Direct Comparison

Upon initiating the test sequence, the binary counter would be set to zero and the MISR (error register) would be initialized to a seed value (zero). Note, since either all 0's or all 1's is written into the memory at any given time, only one data input needs to be generated (all inputs needed to be generated in Scheme 1). Also, the expected output signal (one signal) can be generated from the control bits and address bit A3. The comparison result can then be compressed into an error register.

Advantages

Short test time, good coverage. With direct comparison method, only one expected output line needs to be routed to actual output since only a word of all 0's or 1's is written.

Disadvantages

Mapping from logical to physical addresses due to address scrambling could increase circuitry. Row/column replacement of faulty rows or columns could be a problem to maintain the same fault coverage.

Comments

Dependent on the physical layout of the memory system. MAR faults are not completely detected. Could be extended to handle surrounding PSFs.

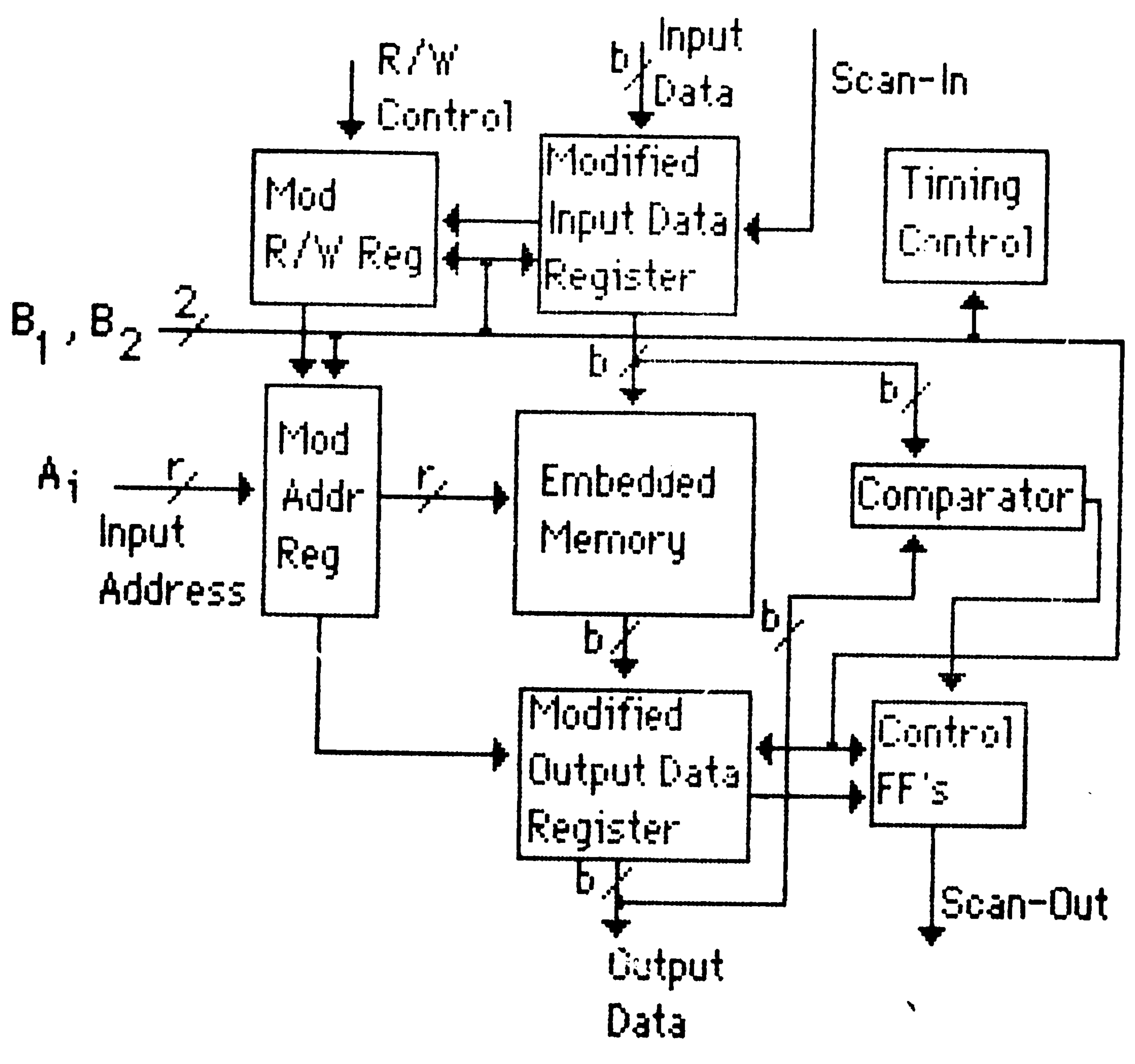
3.3 Sun, Z. & Wang, L: Self-Testing of Embedded RAMs

Overview

This method combines both self-test and scan techniques to test embedded RAMs (See Figure 3.4). The self-test utilizes the Simple March Test Pattern (SMTP) to detect hard failures in the RAM, while the scan technique is used to detect stuck-at and bridging faults in the comparator and data lines, and to diagnose single stuck-at faults in the RAM. The memory is organized n words by b bits wide.

Approach

The Simple March Test Pattern (SMTP) is built into the hardware by modifying the R/W control circuit, the address register, the input data register, and the output data register. The comparison circuit consists of XOR gates between the output register and the complement of the input register which are then Ored together. There are four modes of operation: normal, scan, single-step, and self-test. The single-step mode performs a read or write operation defined by the scanned-in test pattern. The self-test mode executes the SMTP for detecting faults in the embedded RAM array.



B_1	B_2	Operation Mode
0	1	Normal
0	0	Scan
1	1	Single-Step
1	0	Self-Test

Figure 3.4: Block Diagram of Sun & Wang Self-Testing Embedded RAM

Advantages

The SMTP is easily implemented, requires short test time O(5n), physical positions of the cells do not have to map to their logical addresses, scan design allows flexibility.

Disadvantages

Scan technique to aid in self-test is slow (ie; serially scanning each pattern), SMTP is a minimal functional test (See Test Patterns/Algorithms). All the input lines need to be routed to the output lines for comparison.

4. A New Hybrid Approach For Memory BIST

Figure 4.1 shows a new approach using Multiple MISRs (MMISRs) for memory BIST. This method is a hybrid approach that combines direct comparison and data compression techniques. The concept is to logically divide the memory into two equal parts, providing a MISR for each part. Due to the regular structure of a memory system, the resulting signatures in each partition will be the same, assuming that no fault occurs simultaneously in both partitions. Thus, rather than comparing the resulting signature elsewhere, the signatures can be compared to each other. In order to provide high fault coverage with this approach, all the components that are shared between the MISRs must be isolated so that they can be tested independently (ie; the row decoder). Otherwise, an error in a common component could manifest itself into identical errors in each memory partition, and thus, the two MISRs signatures would be the same. To provide the means for isolation, the memory is logically divided into 4 partitions as shown below.

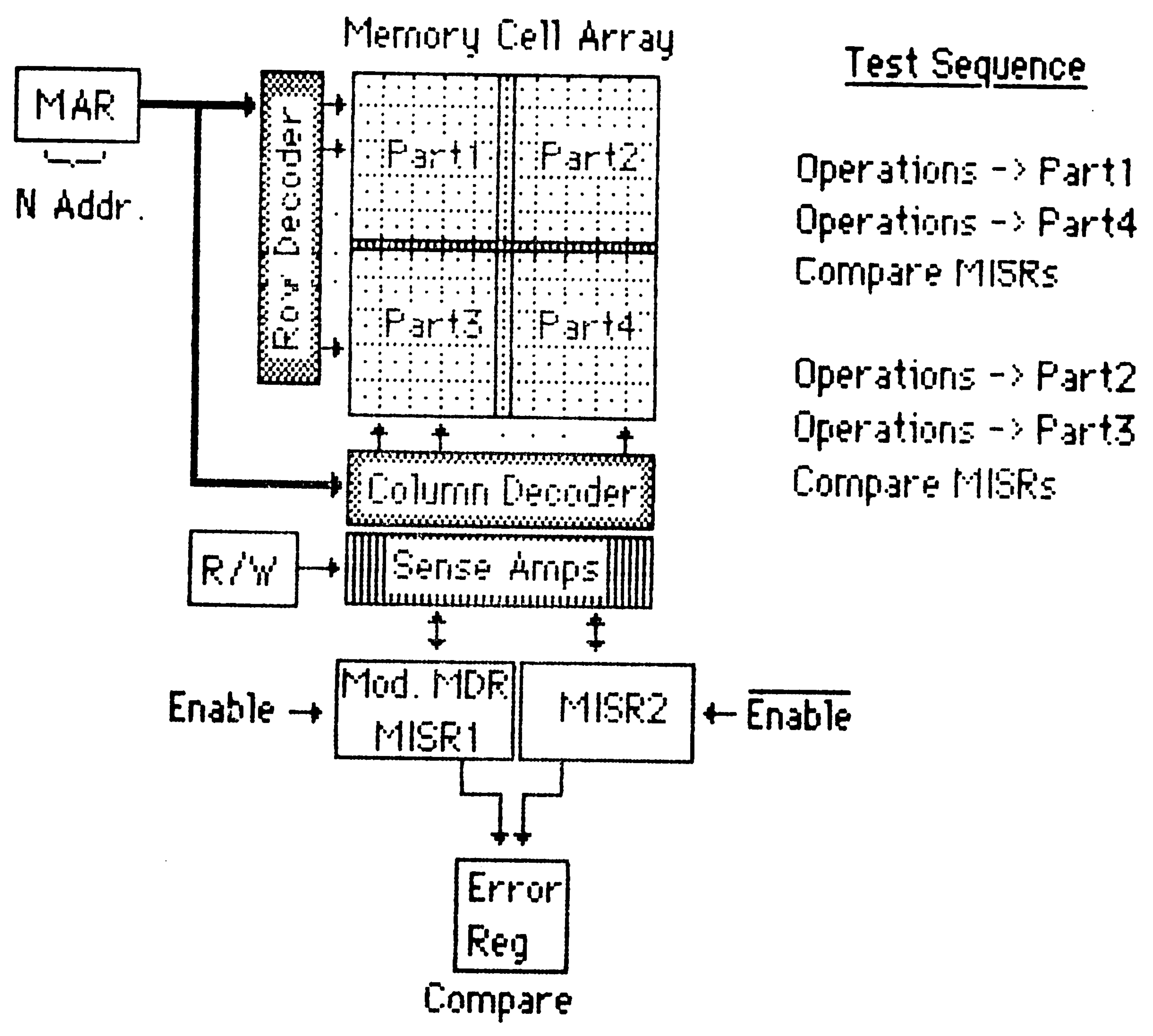


Figure 4.1: A New Hybrid Approach for Memory BIST

The operation of this approach is as follows: The MISRs are only active in a test mode during a read operation. The enable level is high while reading from partitions 1&3

and low when reading from partitions 2&4. Therefore, the MISRs will contain the compressed output data from their associated partitions. Verification of the results is accomplished by shifting and comparing the contents of the MISRs to each other. In order to isolate the row decoder, the test algorithm should be run on partitions 1&4, compare the results, and then run on partitions 2&3 and compare the results. This method of isolation allows faults such as s-a-0 in the row decoder to be detected.

The hybrid approach has the following characteristics:

1) Resulting signatures need not be compared elsewhere, they can be compared to themselves.
2) The comparison can be one line wide by simply shifting and comparing all bits in the MMISRs. This method reduces circuitry but increases time to compare. The comparison could be done in parallel to reduce time, but the added circuitry would increase.
3) Reduction of aliasing can be accomplished by more frequent checking of the signatures.
4) Limited diagnostic capabilities: checking the signatures more often tells when an error occurs; having more than 2 partitions results in a voting system, therefore allowing a faulty partition to be detected and located. Note: the more partitions, the more MISRs, and therefore higher BIST overhead.
5) No additional circuitry for generating expect output data is required.
6) This approach, like the other approaches, can be used with parallel techniques. The parallel operations need to be within each partition.
7) This approach does not fully cover MAR faults (ie; if the MAR only accessed one address).

5. Speed-up Techniques: Parallelism

This section is concerned with speeding up the test application time via parallelism. The concept of parallel testing of memory systems is to internally increase the word size beyond the normal word size while in a test mode. Parallel signature analyzers (PSAs), also known as MISRs, are used to monitor read data in parallel. In addition, PSAs can be loaded with write data to write into the memory in parallel, as long as the signature is shifted out and compared to known results before being loaded with data to avoid losing the signature. (This is unnecessary if there are two PSAs, one for an input data register and one for an output data register or if extra logic is provided to translate the signature to write data without changing the signature register). The approaches suggested are algorithm independent and are described below:

5.1 Han, Sang & Malek, Miroslaw: Two-Dimensional Multiple-Access Testing Technique For Random-Access Memories

Overview

A new type of memory organization using multiple accessing techiques for testing is proposed. This architecture is used to speed-up test time via parallelism while preserving the fault coverage. A k-stage LFSR parallel signature analyzer (PSA) is used to compact read data and can also be used to write k-bits in parallel.

Approach

There are three accessing modes:

Mode-0: A single cell is accessed as in normal operation

Mode-1: k memory cells within one row are selected at a time

Mode-2: k memory cells within k rows are selected at a time

k cells are selected at a time by using multiple select decoders. Mode-2 is restricted to write cycles only. The read data is compressed via a PSA. The data in the PSA can be shifted out for comparison, or data can be shifted in or modified for writing. This allows k cells to be accessed in parallel. The maximum speed-up is $2k^2/(k+1)$, so, the speed-up increases with k, but the larger the k, the larger the PSA required, and thus, the larger the overhead.

Advantages

This approach has the flexibility of using the different modes of accessing within an algorithm. For example, to detect coupling within a k-bit word, set to mode-0 (normal mode) for writing and read using mode-1 (in parallel). The disadvantage with mixing modes is that the speed-up factor is reduced.

Disadvantages

Two pins are required to set modes: normal, mode-1, mode-2, and scan. Due to having multiple write modes, larger line drivers are required to minimize the increase of the access time. Note: this is only applicable in the test mode, since in the normal mode only a single cell is accessed, but nonetheless, having larger line drivers would increase the area overhead.

Comments

The fault coverage in the peripherals (MAR, MDR, and decoder) may be higher without using multiple accessing techniques. This is because with multiple accessing techniques these components are not being tested in their normal mode of operation. For example, a fault in the decoder could occur such that multiple lines are selected but these might have already been selected due to the multiple accessing technique.

6. Design Guide

To summarize the methods and approaches of the BIST schemes discussed earlier, the following individual components of memory BIST are reviewed and analyzed: address generation unit, input data generation unit, and evaluation circuitry. In addition, choosing an algorithm and design goals are discussed. A diagram of memory BIST is shown below:

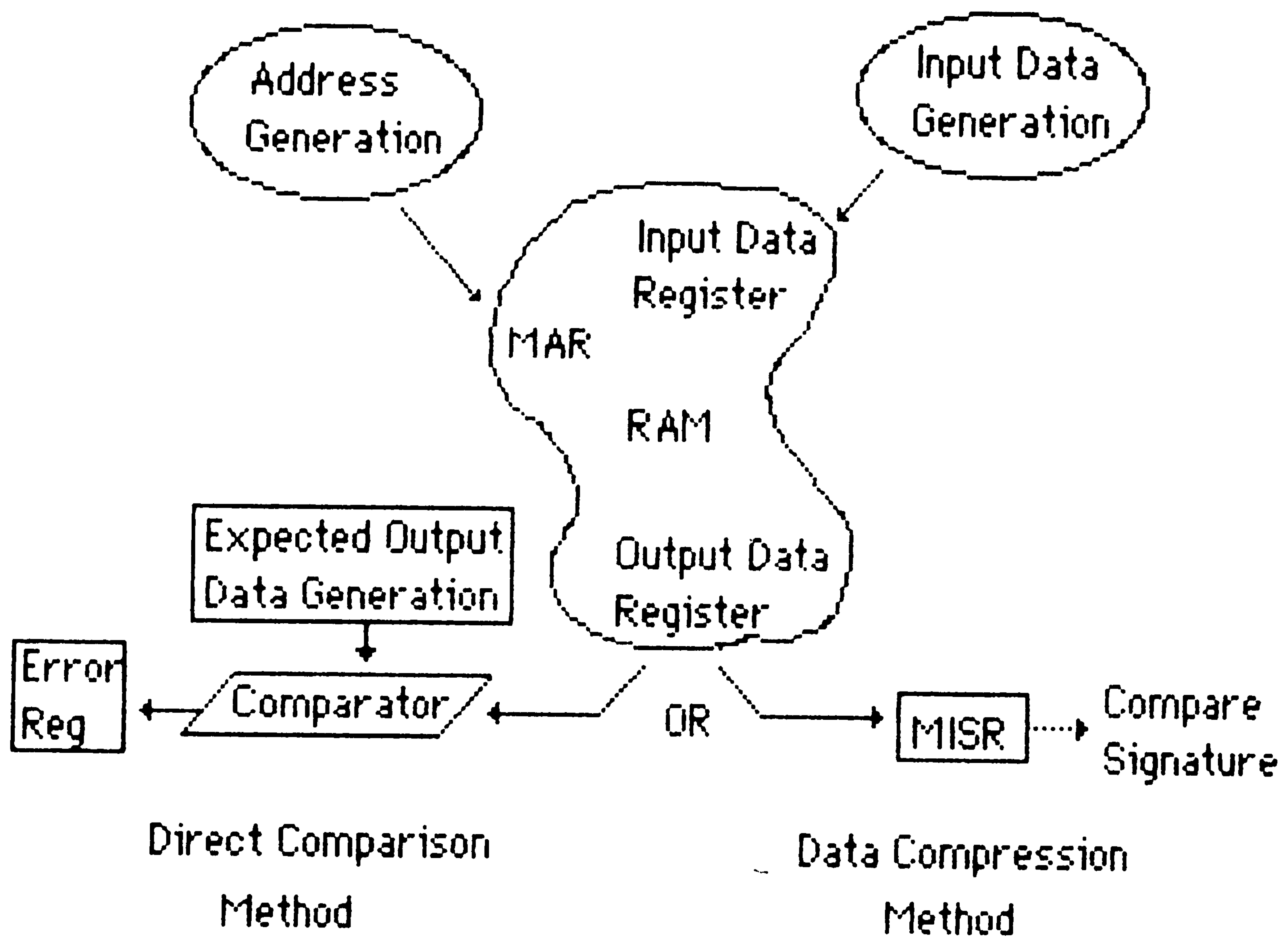


Figure 6.1: Memory BIST

6.1 Address Generation Unit

The address generation unit generates the needed addresses for the algorithm implemented. There are three methods used to generate these addresses: a LFSR, a binary counter, and a translation unit. The LFSR is the simplest hardware to implement. This approach generates pseudo-random addresses (all the addresses are generated, but in a pseudo-random fashion). The LFSR must be modified to include address 0. A binary counter permits addresses to be generated in a sequential manner in either ascending or descending order. A translation unit maps the logical addresses to their physical addresses. The address generation unit should be implemented as a modification to the

MAR with the least amount of extra hardware. Below are the advantages and disadvantages of each approach:

Method	Advantages	Disadavantages
LFSR	Easy to implement	Pseudo-random addresses No way to ascend & descend* No mapping to physical addresses
Binary Counter	Sequ. ascend&descend	No mapping to physical addresses Moderate overhead
Translation Unit	Map to phys. addresses Sequ. ascend&descend	High overhead

* A modified up-down LFSR is presented in [Nicolaidis '85].

6.2 Input Data Generation Unit

The input data generation unit is responsible for loading the input data register (or the MDR) with the correct input according to the implemented algorithm. The hardware to implement the input data generator can be reduced if the algorithm only writes the all 0's or the all 1's word since only one line needs to be used. The advantage of only using one line is the reduction of hardware, but this approach lacks the flexibility of writing different data within the input data register. The input data generation unit should be implemented as a modification to the input data register with the least amount of extra hardware.

6.3 Evaluation Circuitry

The purpose of the evaluation circuitry is to detect an error between the circuit under test and known fault-free results. There are two main approaches for evaluation circuitry: Direct comparison and Data compression.

Direct Comparison

The direct comparison method is accomplished by comparing expected data to the actual output data. This approach has better fault coverage than data compression techniques but may require more circuitry to implement because of the following two reasons: circuitry needed to generate the expected data and the routing circuitry between the expected data and the actual data. One way to reduce the routing circuitry is to have only one line to route: this can be accomplished for certain algorithms where only the all 0's and the all 1's words are written/read, and also for bit organized memories.

Data Compression

The data compression method utilizes either LFSRs or MISRs to compress output data into a signature during the test (refer to Figure 1.6: Data Compression Techniques). When the test is complete, the resulting signature is compared to a known correct signature. Data compression techniques may require less circuitry than direct comparison methods since no circuitry is required to generate expected data output. Also, this technique may have less coverage than direct comparison techniques due to aliasing. Aliasing occurs when an error or errors mask themselves in such a way that the resulting signature is the same as the correct signature: thus the error is not detected. The other problem with this technique is that the signature must be shifted out and compared elsewhere. This requires either ROM or additional test equipment, which should be avoided if possible. Signature analysis does have the advantage of testing itself during data compression. Note: data compression requires two separate data registers (input and output) or some other means to maintain the signature during a write cycle (loading the memory data register and writing into memory).

6.4 Choosing an Algorithm

Analyzing the tradeoffs between algorithms, the following observations are made:

For test complexity of $O(n)$:

1) Test time is not an issue; therefore, parallel approaches need not be implemented.
2) Major concern is to reduce overhead of BIST circuitry.
3) Inherently the fault coverage is not as comprehensive as $O(n^{3/2})$ or $O(n^2)$, but the test times and overhead are small enough for any size memories.

For test complexity of $O(n^{3/2})$:

1) Test times may be acceptable; if not, reduce test time via parallel approaches.
2) Since complex algorithms, major concern is to reduce the BIST overhead.
3) Fault coverage is good.

For test complexity of $O(n^2)$:

1) Test times are unacceptable; therefore, the first concern is to reduce test time via parallel approaches.
2) The next concern is to reduce overhead of BIST circuitry.
3) If BIST overhead is low enough, these algorithms might be well suited for small memories without using speed-up techniques.
4) Fault coverage is good.

In order to cover stuck-at, bridging, transition, destructive read, and coupling faults only $O(n)$ algorithms are required. For pattern sensitive faults (PSFs), $O(n^{3/2})$ or $O(n^2)$ are recommended when mapping of the logical to physical addresses is not available or practical. The test times with these algorithms are too long for large memories, and therefore should be implementated using parallel techniques. In addition, these algorithms are relatively complex, and may require a high BIST area overhead. When knowledge of how the logical addresses map to the physical addresses is available, and the test time requirement is low, then a modified checkerboard approach is recommended.

In choosing an algorithm, having good fault coverage and short test times are important, but the BIST overhead must also be considered. To reduce the BIST overhead and increase the fault coverage at the expense of a small increase in the test time, combining or modifying some of the algorithms previously discussed in section 2.3 may prove fruitful. For example, the march4 algorithm could be modified such that in step 3 an extra R1 is inserted before the W0, and in step 5 an extra R0 inserted before the W1. This modification would increase the fault coverage and reduce the BIST overhead. The reduction of the BIST overhead is due to the symmetry of the modified algorithm which would make the implementation of the control circuitry simpler.

References

[Abadir '83]	Abadir, M. S., Reghbati, H. K., "Functional Testing of Semiconductor Random Access Memories", Computing Surveys, Vol. 15, No. 3, September 1983, PP. 175-198.
[Abraham '85]	Abraham, Bardell, McAnney, Savir, "Built-In Test: Theory and Implementations", 1985.
[Bardell '85]	Bardell, McAnney,"Self-Test of Random Access Memories",1985 IEEE Test Conference,pp. 352-355.
[Bartee '85]	Bartee, Thomas C., Digital Computer Fundamentals, McGraw Hill 1985, pp. 284-291.
[Booth '84]	Booth, Introduction to Computer Engineering Hardware & Software Design, Wiley 1984, pp. 391-394.
[Breuer '76]	Breuer, M. A., Friedman, A.D., Diagnosis & Reliable Design of Digitial Systems, Computer Science Press, 1976, PP. 156-161.
[Daehn '86]	Daehn, W., Gross, J., "A Test Generator IC for Testing Large CMOS-RAMs", 1986 International Test Conference, pp. 18-24.
[Green '86]	Green, C.W., Personal Communication, 1986.
[Jain '86]	Jain, S., Stroud, C., "Built-in Self Testing of Embedded Memories", IEEE Design & Test of Computers, October 1986.
[Jones '86]	Jones, Larry D., Principles & Applications of Digital Electronics, Macmillan 1986, pp. 412-417.

References (Continued)

[Kinoshita '86] Kinoshita, K., Saluja, K., "Built-In Testing of Memory Using an On-Chip Compact Testing Scheme", IEEE Transactions on Computers, Vol. C-35, No. 10, October 1986, pp. 862-870.

[Mukherjee '86] Mukherjee, Introduction to nMos & CMOS VLSI Systems Design, Prentice-Hall 1986, pp. 262-277.

[Nair '78] Nair,Thatte,Abraham,"Efficient Algorithms for Testing Semiconductor RAMs", 1978 International Test Conf., PP. 572-576.

[Nair '79] Nair, R., "Comments on 'An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories'", IEEE Transactions on Computers, Vol. c-28, no. 3, March 1979, PP. 258-261.

[Nicolaidis '85] Nicolaidis, M., "An Efficient Built-In Self Test Scheme For Functional Test of Embedded RAMs", 1985 Fault-Tolerant Computing, pp. 118-123.

[Pradham '86] Pradham, D.K., Fault-Tolerant Computing Theory & Techniques Vol I, Prentice Hall, 1986.

[Sridhar '86] Sridhar, T., "A New Parallel Test Approach for Large Memories", IEEE Design & Test of Computers, August 1986, pp. 15-22.

[Suk '81] Suk, D. S., Reddy, S. M., "A March test for Functional Faults in Semiconductor Random Access Memories", IEEE Transactions on Computers, C-30, 12, December 1981, PP. 982-985.

[Sun '84] Sun, Z.,Wang, L.,"Self-Testing of Embedded Rams", 1984 International Test Conf.,PP. 148-156.

[Westcott '81] Westcott, D. "The Self-Assist Test Approach to Embedded Arrays", 1981 International Test Conf.,PP. 203-207.

References (Continued)

[You '84] You, Y., Hayes, John P., "A Self-Testing Dynamic RAM Chip", _1984 Conference on Advanced Research in VLSI, M.I.T._, January, 1984, PP. 159-168.

Appendix

Memory Fault Simulator

Developed By Steve Lerner

August 1986

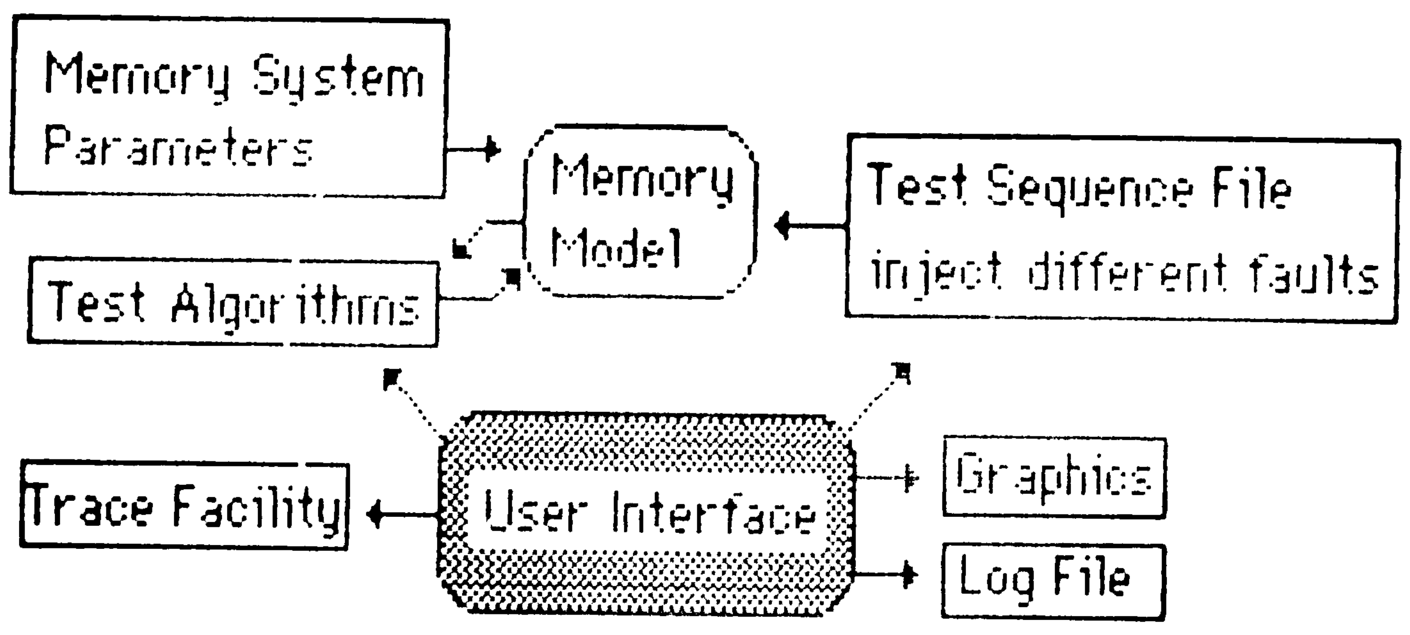
Technical Reference Manual v1.0

User's Guide v1.0

Table of Contents

Overview

The Memory Fault Simulator was developed to aid in the analysis of memory test algorithms used for built-in self test (BIST) of memory systems. Although most of the algorithms used to test memory systems are not that complex, to analyze the fault coverage in terms of memory subsystems (ie; address registers, decoders, data registers, etc.) is much harder. The Memory Fault Simulator has the following features: flexibility of injecting many different types of faults into any of the memory subsystems, user definable memory system parameters, common test algorithms, user definable test algorithms, graphics capabilities, and trace facilities. An overview of the system is shown in the figure below:



Overview of Memory Fault Simulator

The memory system parameters are contained in the MEMPARAM.H file. The standard test sequence file is TESTSEQ.TST, but users may create their own. Each of the test algorithms are stored in their own file. The simulator was developed in C using C-terp*, an interpretive C environment, on an IBM compatible PC. This technical reference manual is divided into the following sections: Memory System Model, Memory System Parameters, Fault Models, Memory Fault Simulator Routines, and Test Algorithm Modules.

* C-terp is a trademark of Gimpel Software

Memory System Model

The Memory Fault Simulator uses a 16 bit memory system with the following architecture:

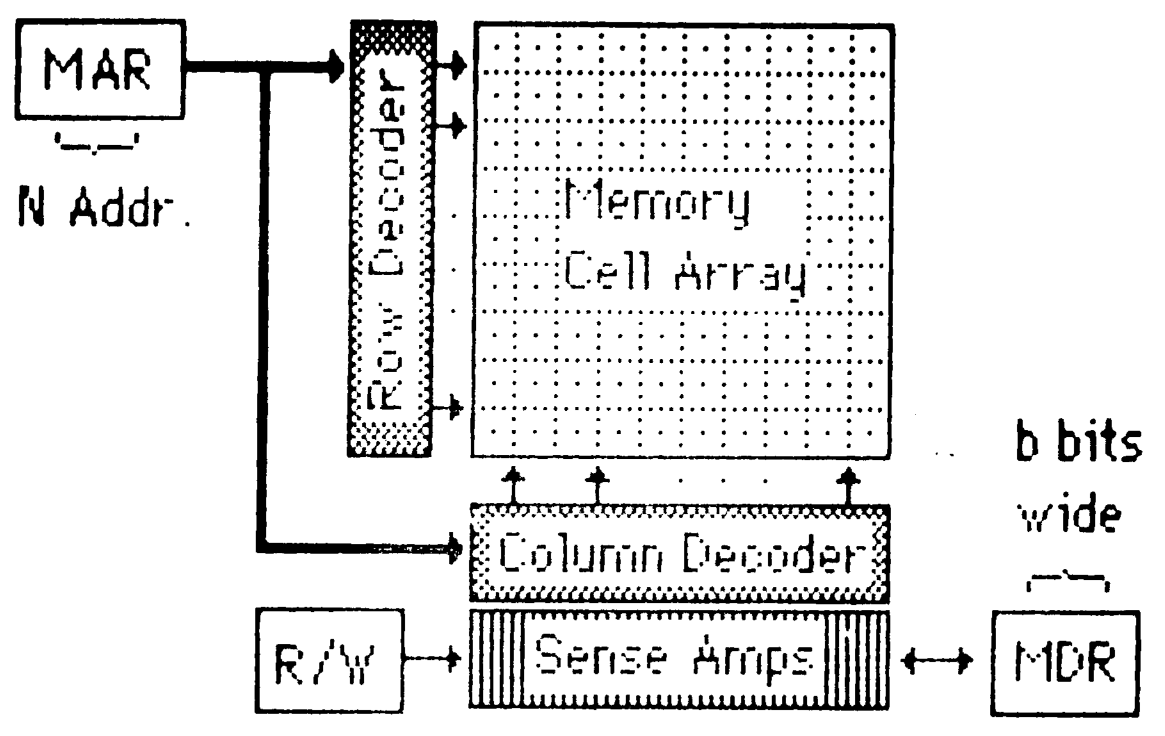


Figure 2: Memory System Architecture

The Memory Address Register (MAR) is 4 bits wide, both the Row and Column Decoders are 2 to 4 line decoders, the Memory Cell Array is 4 X 4, and the Memory Data Register (MDR) is a single bit. For graphics capability, the memory system must be 16 bits organized 4 X 4. Accessing the memory system is accomplished by a subroutine call to memory. The general form is memory(address, R/W, word). Shown below are two examples:

memory(9, "write", 0);	Write a 0 to address 9
readit-memory(5,"read",)	Read address 5, result is returned in readit.

Memory System Parameters

The following memory system parameters are user definable:

Name	Default	Comment
MEMSIZE	16	Size of the memory cell array
MAXSTUCK	3	Maximum number of stuck-ats in cell array
MAXRFAULT	3	Maximum number of destructive read faults
MAXCOUPLED	3	Maximum number of coupled cells in cell array
MAXDECSTUCK	3	Maximum number of stuck-ats in decoders
COLDECSIZE	2	Size of the column decoder (number of bits)
MAXACTIVE	3	Maximum number of decoder lines active at once
MEMANDTYPE	1	1(0) AND-TYPE (OR-TYPE) for memory cell array

Maxstuck, maxrfault, maxcoupled, and maxdecstuck should be as small as possible for increased performance. There is no error checking in terms of the relationship between memsize and coldecsize. These parameters maybe modified and are located in the MEMPARAM.H file.

Fault Models

The common fault models used to test the functionality of a memory system that are implemented in the Memory Fault Simulator are the following: Stuck-at faults, Bridging faults, Destructive Read faults, and Coupling faults. Stuck-at and bridging faults can occur in any of the memory subsystems. Destructive read faults and coupling faults can occur only in the memory cell array. A short description of these fault models follow.

The stuck-at faults model assumes that one or more logic values in a memory system cannot be changed. For example, one or more memory cells could be stuck at 1 or 0.

A destructive read fault assumes that the data in a memory cell is destroyed following a read operation.

Bridging faults are either AND type or OR type. AND(OR) type bridging results in dominant 0(1). For example, if two leads are shorted, in AND(OR) type bridging the value out would be a 0(1) given that one of the leads was a 1(0).

A pair of memory cells (i,j) is coupled if writing a value V into a cell, say cell i, forces cell j to change state. This does not necessarily imply that a similar transition in cell j will influence cell i in the same way.

A list of subroutines that are used to inject faults into a memory system are shown below:

MAR Faults

setmarstuckat0(bits)	Set bits in MAR stuck-at 0 (ie; bits=4, bit 3 s-a-0)
setmarstuckat1(bits)	Set bits in MAR stuck-at 1(ie; bits=5, 1&3 s-a-1)
setmarbridge(kind,bits)	Set MAR AND-Type bridging (kind=A)
setmarbridge(kind,bits)	Set MAR OR-Type bridging (kind=0)

Decoder Faults

setrowdecstuck(line,value)	Set line in row decoder to be stuck-at value (1/0)
setcoldecstuck(line,value)	Set line in column decoder stuck-at value (1/0)

Memory Cell Faults

setstuckat(address,value)	Set memory address to be stuck-at value (1/0)
setreadfault(address,value)	The value in address is destroyed following a read
setcoupled(addr1,addr2, value1,value2)	Address2 is coupled to address1 such that when value1 is written into address1, address 2 is forced to value2

Memory Data Faults

setmdrstuck0(bits)	Set bits in MDR to be stuck-at 0
setmdrstuckat1(bits)	Set bits in MDR to be stuck-at 1
setmdrbridge(kind,bits)	Set MDR AND-Type bridging (kind=A)
setmdrbridge(kind,bits)	Set MDR OR-Type bridging (kind=O)

Valid parameters are	bits	A decimal number used to represent bits ie; 5 would mean bits 1 & 3
	kind	Either A for AND Type, or O for OR Type
	line	Refers to lines of a decoder: range 1 to 2^n lines, where n=# bits in the decoder. ie; 2 to 4 line decoder has lines 1 through 4
	value	Either 1 or 0
	address	Currently only addresses 0-9 are available. See Limitations.

For examples of how to set faults within a memory system, refer to the Memory Fault Simulator User's Guide section on Creating Test Sequence Files.

Memory Fault Simulator Routines

A listing of each of the routines in the Memory Fault Simulator follow:

Name	Parameters	Description
initialize	()	Initialize all variables except memory
	---------------------------- Set Faults in Memory	System ----------------------------------
setmarstuck0	(bits)	Set stuck-at 0 in MAR
setmarstuck1	(bits)	Set stuck-at 1 in MAR
setmarbridge	(kind,bits)	Set bridging faults in MAR
setrowdecstuck	(line,value)	Set row decoder stuck-at
setcoldecstuck	(line,value)	Set column decoder stuck-at
setstuckat	(address,value)	Set stuck-at in memory cell array
setreadfault	(address,value)	Set destructive read faults in memory
setcoupled	(addr1,addr2,value1,value2)	Set coupling faults in memory cell array
setmdrstuck0	(bits)	Set stuck-at 0 in MDR
setmdrstuck1	(bits)	Set stuck-at 1 in MDR
setmdrbridge	(kind,bits)	Set bridging faults in MDR

Name	Parameters	Description
---------------------------- Simulate Memory System with Faults ----------------------------		
memory	(address,rw,word)	Simulate memory system with faults
---------------------------------- Memory System Faults ----------------------------------		
marfault	(address)	Return address under faulty condition
mdrfault	(outdata)	Return data under faulty condition
decfault	(decoder,line)	Setup decoders under faulty condition
stuck	(addr)	Returns 1 if addr is stuck
coupledwrite	(address,value)	If coupled fault, write to coupled cell
------------------------------------- Error Routine -------------------------------------		
errmsg	(addr)	Displays error message
------------------------------------ Display Routines ------------------------------------		
display	(r,c,value)	Displays on 4 X 4 grid
initdisplay	()	Initializes 4 X 4 gid display
---------------------------------- Library Routines ----------------------------------		
getline	(s,lim,fname)	Reads line from specify filename
index	(s,t)	Returns index of t in s, -1 if none
strcopy	(s,t)	Copies t to s
strcmp	(s,t)	Return <0 if s<t, 0 if s=t, >0 if s>t

<u>Test Algorithm Modules</u>

Currently, there are eleven algorithms implemented in the Memory Fault Simulator: MSCAN, Checkerboard, SMTP, 4 March test patterns, Marinescu, Nair, Walking 1's and 0's, and GALPAT. Each algorithm is contained in its own file, and references are made to the memory model by memory system parameters and the following routines: memory and errmsg. Each algorithm is a routine written in C and is currently included in the Memory Fault Simulator. Below is an example of the SMTP test algorithm written in pseudo-code and in C:

Pseudo-Code

```
smtp

Initialize memory to 0
For all addresses
  R0
  W1
For all addresses
  R1
  W0
```

C

```
 smtp()
{int i;
 for (i=0;i<MEMSIZE;i++) memory(i,"write",0);
 for (i=0;i<MEMSIZE;i++) {
   if (memory(i,"read",)!=0) {errmsg(i),return}
      memory(i,"write",1)}
 for (i=0;i<MEMSIZE;i++) {
   if (memory(i,"read",)!=1) {errmsg(i),return}
      memory(i,"write",0)}
 }
```

Each test algorithm subroutine must have an unique name. In addition, to implement new algorithms, the following needs to be modified in the Memory Fault Simulator: In the user interface, under available test algorithms, include the algorithm name; In the section that runs the algorithm, modify the compound if statement to include the new algorithm, ie; add: else if (strcmp(algo,"newname") == 0) newname(), this tells the simulator to run the new algorithm if selected; In the section where algorithms are included, add #include "filename" to tell the simulator where the algorithm is stored.

Limitations

The major limitation of the Memory Fault Simulator is the lack of speed and the fault models. Future recommendations to fault models would be to include hold faults, transition faults, and pattern sensitive faults. Currently there is a limitation when setting faults using the memory test sequence file: due to the Memory Fault Simulator interpreter, only addresses 0-9 can be used in setting faults such as stuck-at, destructive read, and coupling within the memory cell array. Also bridging faults in the decoders and the memory cell array are not yet available.

Getting Started

The Memory Fault Simulator was developed using C-terp*, and therefore, it is recommended to use C-terp when running the simulator. C-terp is an interpretive C environment with a full screen text editor and debugging facilities. To get started, the files ct.exe and stdio.h as well as all the Memory Fault Simulator files should be copied into a convenient directory. C-terp should then be executed by typing "ct". The following menu should be displayed:

```
                C-terp

        Compile             Pre-Process
        Edit                Quit
        File list           Run
        Global search       System
        Load                Unload
        Options             Write

        Command:
```

Now, load in the Memory Fault Simulator by typing "L". The user will be prompted for a file name. Type in "mem.c" and press carriage return <CR>. To run the Memory Fault Simulator, press "R" followed by a <CR>. All the files that the simulator require will automatically be loaded. The Memory Fault Simulator should now be running. After an algorithm is run, C-terp will ask the user to press any key. After pressing a key, the main C-terp menu will be displayed (see above). To quit, simply press "Q". To run the simulator again, press "R" followed by a <CR>.

The C-terp editor will allow the user to easily create and modify test sequence files, algorithms, etc. To use the on-line help, press the alt key and the "h" key simultaneously.

Refer to the C-terp reference manual for more details on using C-terp and its facilities.

* **C-terp is a trademark of Gimpel Software.**

The Memory Fault Simulator

The user interface to the Memory Fault Simulator is a menu driven system which prompts the user for needed information to run the simulator. Shown below are the menu system prompts with explanations (italics are default values):

```
                Analysis of Memory Test Algorithms

        Enter Test Sequence Filename: testseq.tst
        Enter Test Algorithm Name   :
        Store Results in <filename> : algotest.dat
        List Each Transaction       : no
        Display Graphically         : no

                Is Input Correct : yes
```

The test sequence file consists of a list of tests using different types of injected faults. For example, one may wish to run a series of tests on the memory cell array with different faults: ie; stuck-at-0, stuck-at-1, coupling, etc. See Creating Test Sequence Files.

The test algorithm name is the name of the algorithm that is to be run. If a carriage return <CR> is pressed, a list of the available algorithms with their test complexity will be displayed like the following:

```
Available algorithms:  mscan,  checker,  smtp,   marcha,  marchb,  march,   march2,
                       0(4n)   0(4n)     0(4n)   0(9n)    0(11n)   0(14n)   0(15n)

                       mari,   nair,     walk 1's,  galpat
                       0(17n)  0(30n)    0(n^2)     0(n^2)
```

The results of the simulation is stored in a log file. The form of the filename is filename.ext. The maximum length allowed is 15 characters. The file will be stored in the current default directory.

Listing each transaction is equivalent to a trace facility. Every transaction to the memory model within the algorithm will be displayed on the screen and listed in the log file.

The graphic display allows the user to "see" the algorithm running. A 4X4 grid is displayed along with either the name of the test running or each transaction listed depending on what option was selected.

The user has the chance to verify the input. If an error is made, the user can type no to the prompt, and the menu system will start over. Otherwise, upon correct input, the Memory Fault Simulator will begin.

Creating Test Sequence Files

Test sequence files allows the user to run many tests using the same algorithm with many different types of injected faults. The general form of a test sequence file is shown below:

```
/---------------- Form of Test Sequence File -------------/
Test 1A Dummy
  setstuckat(2,0);
End of Test

Test 1B Stuck-at-1
  setstuckat(2,1);
End of Test
/---------------------------------------------------------/
```

A test begins with the word Test and ends with End of Test. The line beginning with the word Test will be displayed on the screen during execution. There can be different type of faults as well as multiple faults injected into the memory model within each test. Comments are enclosed between /- and -/. Blank lines are also allowed. Refer to the Memory Fault Simulator Technical Reference section on Fault Models for a complete list of the available routines used to inject faults into the memory model.

Creating the test sequence file should be done with a text editor and then saved to disk. It is recommended that C-terp is used. The test sequence file can then be accessed in the Memory Fault Simulator via the menu system prompt: Enter Test Sequence Filename.

Modifying Memory System Parameters

Refer to the Memory Fault Simulator Technical Reference section on Memory System Parameters for a complete list of user modifiable memory system parameters. To actually modify these parameters, one needs to edit the MEMPARAM.H file, make the modifications, and save the results. Then, the Memory Fault Simulator should be loaded and executed via C-terp.

Adding New Test Algorithms

Refer to the Memory Fault Simulator Technical Reference section on Test Algorithm Modules.

<u>**Examples**</u>

Sample Session

```
                 Analysis of Memory Test Algorithms

Enter Test Sequence Filename: sampleseq.tst
Enter Test Algorithm Name   : smtp
Store Results in <filename> : sample.log
List Each Transaction       : no
Display Graphically         : yes

             Is Input Correct : yes


. . . . . . . . . . . . . Testing Memory . . . . . . . . . . . . .

                           ____________________
                           |   |   |   |   |   |
                           |___|___|___|___|___|
                           |   |   |   |   |   |
                           |___|___|___|___|___|
                           |   |   |   |   |   |
                           |___|___|___|___|___|
                           |   |   |   |   |   |
                           |___|___|___|___|___|
```

Test 1 Sample Test

Sample Test Algorithm

Below is the Simple March Test Pattern (SMTP) in both pseudo-code and in C.

```
Pseudo-Code                    C

smtp                           smtp()
                               {int i;
Initialize memory to 0          for (i-0;i<MEMSIZE;i++) memory(i,"write",0);
For all addresses               for (i=0;i<MEMSIZE;i++) {
  R0                               if (memory(i,"read",)!-0) {errmsg(i),return}
  W1                                  memory(i,"write",1)}
For all addresses               for (i-0;i<MEMSIZE;i++) {
  R1                               if (memory(i,"read",)!=1) {errmsg(i),return}
  W0                                  memory(i,"write",0)}
                               }
```

Sample Test Sequence

```
/--------------------Sample Test Sequence-------------------/
Test 1 Sample Test
  setstuckat(5,0);
End of Test

Test 2 Sample2 Test
  setcoupled(2,3,0,1);
  setstuckat(5,1);
End of Test
/-----------------------------------------------------------/
```

Sample Log File

```
********************** Testing smtp Algorithm ******************
/--------------------Sample Test Sequence-------------------/
Test 1 Sample Test
  -> Memory Error Detected in Address: 5 <-

Test 2 Sample2 Test
  -> Memory Error Detected in Address: 3 <-
/-----------------------------------------------------------/
***************** End Of Test For smtp Algorithm *****************
```

Sample Trace File

```
********************** Testing smtp Algorithm ******************
/--------------------Sample Test Sequence-------------------/
Test 1 Sample Test

Writing 0 to memory loc 1,1  Address: 1  Logical Address: 0
Writing 0 to memory loc 1,2  Address: 2  Logical Address: 1
Writing 0 to memory loc 1,3  Address: 3  Logical Address: 2
                         .
                         .
                         .
Read 0 from memory loc 2,1 Address: 5 Logical Address: 4
  -> Memory Error Detected in Address: 5 <-
                         .
                         .
                         .
/-----------------------------------------------------------/
***************** End Of Test For smtp Algorithm *****************
```

Biography

Steve Lerner was born in Lexington, Massachusetts in 1961. He graduated from Lexington High School in 1979 and received his BS from Lehigh University in Computer Engineering in 1983. He worked for Texas Instruments as an Electrical Design Engineer/Systems Analyst. Steve anticipates receiving his MS in Electrical Engineering from Lehigh University as of January 1987.