Enabling Embedded Memory Diagnosis via Test Response Compression

John T Chen Janusz Rajski* Jitendra Khare**[1] Omar Kebichi* Wojciech Maly

Carnegie Mellon University - Department of ECE
jtchen@ece.cmu.edu
*Mentor Graphics Corporation - Design for Test Group
**Intel Corporation - Sacramento - DFT/DFM Group

Abstract

This paper introduces a method that enables failure diagnosis of BISTed memories by compression of test responses. This method has been tested by simulation of memories with various specifications, fail patterns and test algorithms. The proposed method has been implemented in 0.18µ CMOS IC.

Keywords: BIST, diagnosis, RAM testing, bitmap, process monitoring, memory repair.

1 Introduction

As the feature size of the CMOS process continues to shrink and the number and size of embedded memories continues to increase, the ability to diagnose embedded memory failures becomes more important.

Such a diagnosis serves two main purposes. First, if the coordinate of failing memory cells can be located (or even just approximated), its failing memory may be repaired by replacing the failing cells via laser-fuse blowing. Second, the type and location of the fault can be analyzed to provide feedback to improve manufacturing yield [1][2].

Embedded memories are inherently difficult to test externally because of their limited controllability and observability. Thus, Built-in Self Test (BIST) has become a popular method to test such memories. The BIST circuit, consuming a small amount of silicon area, generates test inputs for the embedded memories; thus eliminating the need for extra input pins. The BIST circuit can either compress the test response into a "signature" to be scanned out at the end of test execution, or it can check the correctness of the test response and output a "PASS/FAIL" signal at the end of the test execution. Either way, the requirement for output pins is reduced since the complete test response is not exported off the chip. The problem is that neither the signature nor the PASS/FAIL signal is sufficient to diagnose the memory.

Several methodologies exist that attempt to address the above problem. First, additional probe pads are added to the wafer for diagnosis, but probe-pads result in a substantial increase in the circuit area and become useless after packaging. Furthermore, when a fail is detected, the test execution is halted to scan out register values[3][4]. If fails occur frequently, the test time is long.

In [5], the BIST circuit is proposed to record the location of failing cells during testing. However, this hardware will capture only certain types of fail patterns (single cells and columns) and does not record in which step of the test execution the cells failed.

This paper introduces a novel method to enable diagnosis of embedded memory by compressing the test response on-chip to reduce the I/O pins needed. This method consumes a moderate overhead area, does not increase test time, and can identify a high variety of fail patterns. Because static random access memories (SRAMs) are the dominant embedded RAMs used today, this discussion will focus mainly on SRAMs. This compression methodology may be applicable on DRAMs as well.

The rest of the paper is organized in the following fashion: First, Section 2 outlines common memory testing and diagnostic practices. In Section 3, the methodology to enable diagnosis will be proposed. Section 4 shows the simulation results of applying the proposed methodology on a variety of memory specifications, test algorithms and fail patterns. In Section 5, the hardware implementation of the methodology will be presented.

2 Memory Testing and Diagnosis

Fig. 1 shows a typical example of memory BIST. The shaded area represents the BIST circuit. During test execution, the BIST circuit generates the input vectors, including address, data-in, write enable and others. At each read operation, the value at the data-out bus is compared with its expected value, also provided by the BIST circuit. The out-

1. Now at Ample Communications

put of the comparison module, *fail*, becomes *1* if its memory output does not match its expected value, otherwise *fail* becomes *0*.

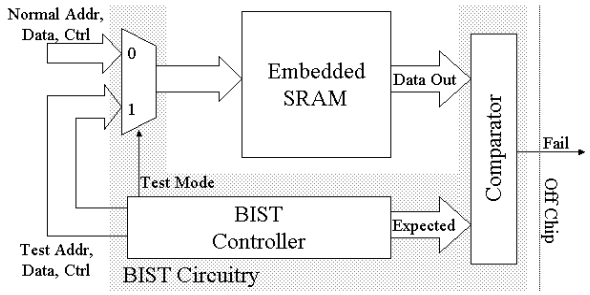

Figure 1. Memory BIST

2.1 Memory Bitmap

During external testing, a test response can be collected to generate a memory bitmap (Bitmaping is a vehicle of diagnosis in which each bitmap displays the location and behavior of the failing memory cells of a memory array). For manufacturing feedback, the bitmap may be classified into different *fail patterns*. Using results from failure analysis or fabrication simulation, a dictionary can be created mapping the fail patterns to defects, which is a very powerful manufacturing process monitoring tool. For example, when a yield problem occurs, the observed fail patterns may help to narrow down the suspected manufacturing steps.[6]

Several example of *monochrome* (the white area represents good cells and black squares represent failing cells) bitmap patterns are shown in Fig. 2: Aside from a single, failing cell (a), the failing cell may form a cluster (also shown in a). The failing cells may also form a column (b) or a row (c). Further, part of a row (or column) may be failing (d), and a combination of a column and a row of failing cells may also occur (e). Finally, two rows (or columns) may fail as shown in (f).

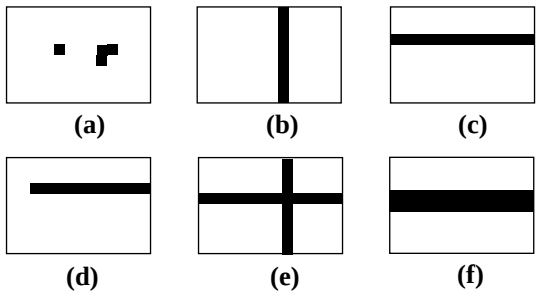

Figure 2. Example Bitmaps

The above list of seven fail patterns by no means represents an exhaustive collection. Obviously, the more fail patterns, the higher the diagnostic resolution [2]. A large number of fail patterns can be collected from a large number of silicon samples or Monte-Carlo simulation results. Color bitmaps, instead of monochrome bitmaps, will be needed to differentiate fail patterns by indicating failing behavior of the cells with different colors or shades. In the simplest case, a cell may be stuck at a hard 0 or 1 or the inverse of the expected value, but a cell may also fail only at specific read cycles and pass others. Diagnostic resolution can only be as good as the test response used to compile the bitmap. But, the band-width needed to obtain the test response for bitmap generation may require unacceptable overhead of I/O pads if traditional memory BIST is applied.

3 Proposed Compression Methods

To overcome the limitations of the traditional memory BIST technique, the following characteristics of the test methodology must be met:

- Low pin overhead - Modern designs are often more severely pin-limited than area limited;
- Ability to generate color bitmaps;
- The ability to be tested at-speed - Many faults may be detected only with at-speed tests;
- Enabling of production test - The time required should be short enough for the production test;
- Scalability - To handle memories of different dimensions;
- Compatibility with existing BIST hardware;
- Handling most production test-algorithms[7];
- Moderate area overhead - As minimum feature size continues to decrease, modest area overhead has become increasingly more acceptable.

As shown above, the proposed solution is targeted for modern designs in which I/O pins are very costly. This paper describes an attempt to design memory BIST which meets the above requirements.

3.1 Reduction of Test Response Bandwidth

In order to reduce the pin requirement for diagnosis, the proposed method compresses the test-responses before exporting them off-chip via I/O pins. The test response may be later restored by decompressing the stored data by a software program on a workstation.

The test response to be compressed, otherwise known as the *fail vector*, is the bit-wise result of comparing the memory output with the expected output, as shown in Fig. 3. The width of the fail vector is inherently equal to the word width of the memory, denoted as *m*. A *0* in the fail vector indicates that the corresponding bit memory output matches its expected value; a 1 indicates a mismatch. The *fail* signal can still be obtained by applying an OR operation to each fail vector.

The proposed compression algorithm generates a six-bit output for each fail-vector input. The effective compression

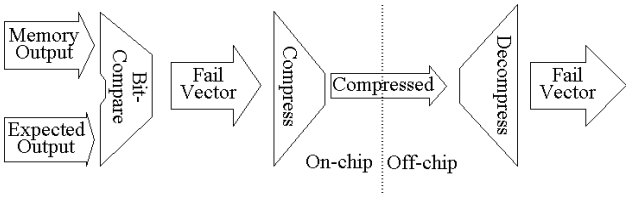

Figure 3. Compression/Decompression Data Flow

ratio is the size of each fail-vector divided by six. For example, if the fail-vector contains 32 bits, the compression ratio is approximately 5.3 (32/6).

3.2 Compression Algorithm

Subsequent rows of fail-vectors generated from a test execution composes a *fail matrix*, as shown in Fig. 4. The fail matrix is the test response to be compressed and decompressed; it is also what is needed to generate a bitmap. In the compression algorithm, we use three types of subsets of the fail matrix: row, column segment or diagonal segment.

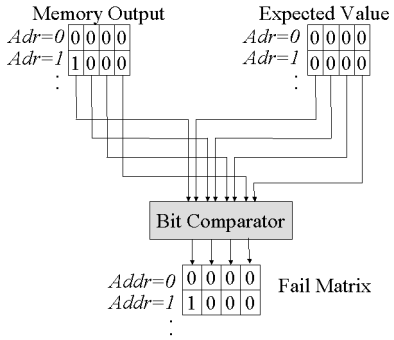

Figure 4. Fail Matrix

3.2.1 Rows and functions applied

The six-bit compressed fail vector is created by applying functions to elements of each subset. The left side of Fig. 5 shows an example of a fail matrix with its rows inscribed in ovals. The three functions applied are the AND, OR and 2OR (See the example shown in Fig. 5). The AND function results in a *1* if the row contains all *1*s, and the OR function results in a *0* if the row contains all *0*s. The AND and OR functions are chosen because they are able to describe whether the rows are all *1*s or all *0*s, both of which occur frequently. The 2OR function results in a *1* if the row contains 2 or more *1*s; otherwise the 2OR function results in a *0*. The decision to use the 2OR function resulted from simulation experiences.

The application of these three functions on a row may produce only four possible permutations. The four rows in Fig. 5 illustrate four such permutations. These four permutations may be arbitrarily encoded into two bits, as displayed

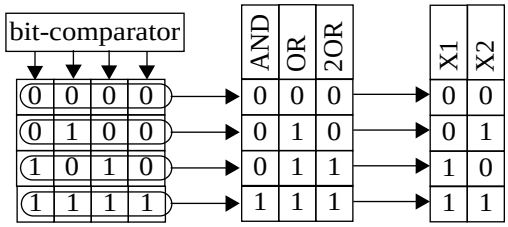

Figure 5. Functions for rows

on the right side of Fig. 5. Thus, the rows produce two of the six compressor output bits.

3.2.2 Column segments and functions applied

In our algorithm, the left-most column is fragmented into segments of *m* (*word width*, Section 3.1) elements. These segments of *m* elements and the remainder *m-1* or less elements are the column segments of the left-most column. Then, each subsequent column is segmented in the same fashion with its segmentation alignment shifted down by one element with respect to the previous column. The left hand side of Fig. 6 (a) shows an example of a fail matrix with its column segments inscribed by ovals. The MASKED AND, MASKED OR and REPEAT functions are computed for each column segment.

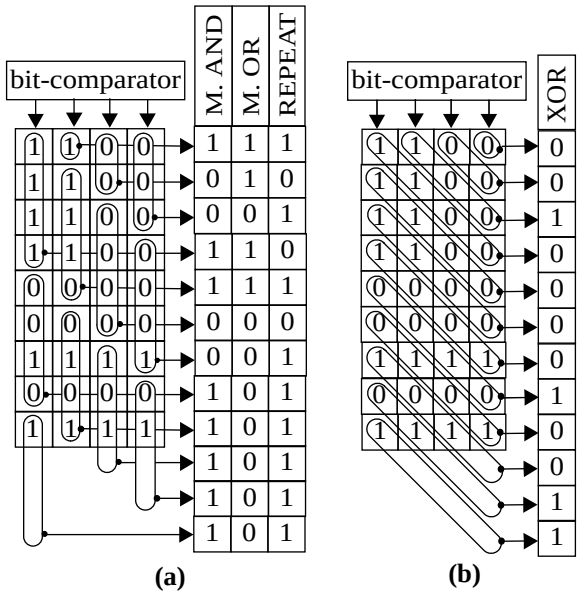

Figure 6. Functions applied to column segments and diagonal segments

The MASKED AND, similar to the AND function, results in a 1 if all elements within the column segment are *1*s. For example, the fourth result of the AND function in Fig. 6(a) is a *1*. The MASKED AND differs from the regular AND function such that any *0* in the column segment is ignored, or masked, if that *0* also belongs to an all-*0* row.

Masking helps to increase the probability that the AND function will encode a column segment of *1*s even if it contains *0*s. In Fig. 6(a), the fifth result of the MASKED AND function is a *1* although the column segment contains a *0* because the *0* is masked since it belongs to an all-0 row. The MASKED AND function is used because of its ability to encode a column segment of *1*s, which occurs frequently in a fail matrix.

Conversely, the MASKED OR is similar to the OR, except for the fact that any *1* within the column segment that belongs to a row of all-*1*s, will be masked from the function. For example, the seventh OR result in Fig. 6(a) is a 0 although the corresponding column segment contains a *1*. The MASKED OR function is applied to the column segment because it can encode series of *0*s.

The REPEAT function results in a 1 if all elements in the column segment are identical to the elements to their immediate left (elements in the left-most column are compared to the elements in the right-most column). Otherwise, the REPEAT function produces a 0. The REPEAT function takes advantage of the fact that neighboring columns of the fail matrices are often similar. No masking occurs for the REPEAT function.

The column segment produces three of the six compressor output bits.

3.2.3 Diagonal segments and function applied

The diagonal segments are identified by grouping elements with their top-left and bottom-right neighbors. The example on the left hand side of Fig. 6(b) shows diagonal segments in ovals. An XOR function is applied to the elements within each diagonal segment and produces a 1 if the number of 1s is odd, a 0 if even. The XOR function helps to encode elements that were not encoded by AND, OR, or REPEAT in other subsets. The diagonal segment produces one of the six compressor output bits.

3.3 Software Decompression

The fail matrix is reconstructed in an iterative fashion. First, a matrix identical in dimension to the original fail matrix is initialized with all its elements marked as unknowns, denoted as *x*s. The *x*s are iteratively replaced by *0*s and *1*s until replacement of any *x* is no longer possible.

3.3.1 Determination Rules

The *x*s are replaced with *0*s *and 1*s using the *determination rules* shown in *Fig. 7.* Each of the five functions described above (AND, OR, 2OR, REPEAT, and XOR), regardless of the subset they are applied to, is associated with two determination rules; one applies if the function results in a *1*, and the other one applies if the function results in a *0*. For example, *Fig. 7 (a) and (b)* illustrate the determination rules for the AND operation. If the result of an AND function applied to a set of elements is a *1*, then it may safely be assumed that all elements within the set are *1*s. Conversely, if the result of an AND function applied to a set of elements is a *0*, it can be assumed that at least one *0* exists in the set; if all but one element has already been assigned a *1*, a *0* may be assigned to the only undetermined element. The process of applying determination rules to a group of elements is referred to as *determination.*

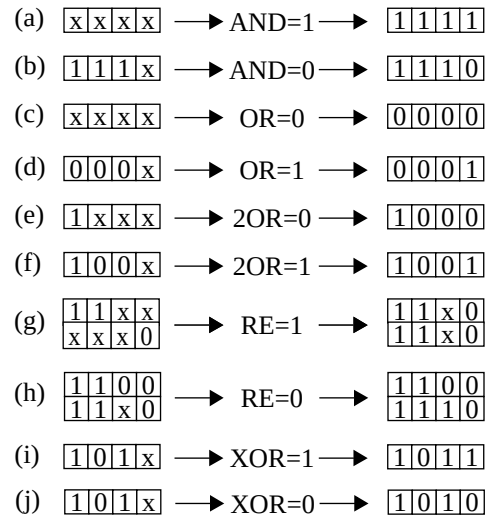

Figure 7. Determination rules

The determination rules for AND and OR can be applied to MASKED AND and MASKED OR by ignoring the masked elements.

Determination rules (a) and (c) are *independent determination rules* because they do not depend on any previously determined elements; the other rules are called *dependent determination rules*.

3.3.2 Determination Scheduling

Determination that leads to the assignment of a *1* or *0*, overwriting an *x*, is considered a *successful determination*. Since independent determinations do not depend on any previous assignment, they should be applied to all subsets first, and they do not need to be re-applied. On the other hand, any time an *x* is removed, there exists a chance for a new successful determination in all subsets that contain this element by a dependent determination rule. Thus, dependent determination rules need to be applied iteratively until either all *x*s have been removed, or no new successful determination is possible if *x*s still exist. If all *x*s are removed, the decompressed fail matrix is identical to the original fail matrix.

How the dependent determination rules are applied to reach the point when no *x* can be removed dramatically influ-

ence the run-time of decompression. The value of an unknown element can often be determined by different rules and from different subsets. Some determination rules, if successful, are able to determine many unknown elements, such as REPEAT=1, while others may determine only one unknown element, such as XOR=1. Also, some determination rules have low probability of successful determination, such as AND=0. The key to speedy decompression is to apply the determination rules at the right time, in the right order and to the right subsets. Two techniques to efficiently apply the dependent determination rules are shown below:

First, the determinations of all subsets that cross a newly assigned element may be queued for execution as a *determination task*. The order in which queued determination tasks may be executed is determined by the effectiveness of the determination tasks, which can be learned by trial and error. New successful determination is not possible when the determination queue has been depleted.

Another method to coordinate dependent determinations is to use windows. After independent determination rules have been applied to all subsets, *x*s typically cluster and occupy small regions of the matrix. Windows are created to enclose these regions. The dependent determination rules are applied repeatedly to every subset that overlaps with the windows repeatedly until no successful determination occurs after applying a pass of all the determination rules. The windows may be updated as the *x*s are being removed.

In either approach, bookkeeping of the subsets will reduce decompression time significantly by minimizing traversals to the elements. For example, the XOR=0 determination rule applied to a diagonal subset will lead to successful determination only if the subset contains one *x*. If the number of *x*s of a subset is recorded and updated, the program will not need to visit every element of the subset to determine the number of *x*s each time the XOR determination rule is applied.

The decompression run-time has a linear relationship with the size of the fail matrix. Also, because of the iterative nature of the decompression algorithm, the run-time varies depending on the fail pattern. In our experience, decompression of 100 memories of the specification 256 words x 32-bit words, tested by March C- algorithm, consumes approximately 32 seconds on a 300 MHz UltraSparc II work station.

3.4 Multiple-Memory Diagnosis

The memories of interest for diagnosis may be tested by BIST either in series or in parallel. If those memories are tested in series, a single compressor may be used to compress the test response of different memories. The compressor may need to be designed to handle the different word-widths of different memories.

If the memories to be diagnosed are tested in parallel, a dedicated compressor for each memory or a shared compressor for all memories may be used. With dedicated compressors, in addition to an area overhead, additional I/O pads are needed because each compressor will generate its own compressed test response. However, this may be advantageous for interconnect-limited designs as fail-vectors do not need to be routed to a single compressor.

For memories to share a single compressor, the fail vectors of different memories can be multiplexed to the compressor. The faulty fail vector is selected to be compressed.

4 Software Simulation

Experiments to determine the quality of this scheme are carried out by software simulation. Fig. 8 details the simulation flow of one simulation run. First, a bitmap is generated from the fail pattern of interest. By emulating the test execution, a fail matrix may be obtained from the bitmap. Using the algorithm described in Section 3, the fail matrix is first compressed into a six-bit wide *compressed fail matrix*, then decompressed to a *decompressed fail matrix*. If all *x*s have been removed, the decompressed fail matrix is considered to be *fully reconstructed*, else *partially reconstructed.*

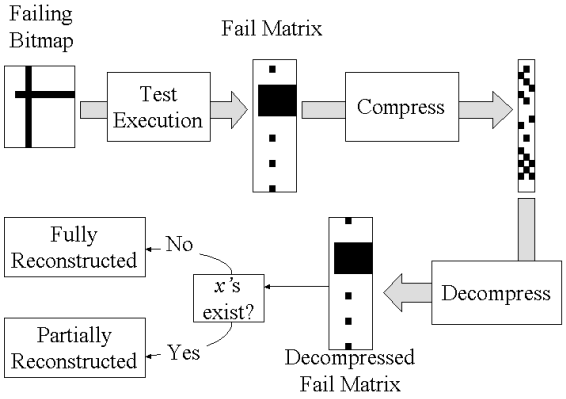

Figure 8. Simulation Data Flow

4.1 Fail Patterns

A total of 108 fail patterns are used in this experiment. It is not possible to describe all the fail patterns in detail. However, they are categorized into 17 classes as shown in Table 1. The integer in the parenthesis after each class represents the number of fail patterns in that class.

Although most of the fail patterns used are static faults, this method can also work well for dynamic faults, such as transition faults and pattern sensitive faults, because they produce the similar geometric patterns that this algorithm is designed to encode. Other, and possibly more complicated, fail patterns may occur in a bitmap. However, for diagnosis purposes, it is sufficient to be able to identify a majority of the fail patterns.

During the simulation, one hundred bitmaps are generated for each fail pattern. For example, a fail pattern may

Table 1. Fail pattern classes

1. Single Cell Stuck At (3)
2. 2 Neighboring Cells Stuck-At (6)
3. 3 Cells Stuck-At (8)
4. 2x2 Cells Stuck-At (7)
5. Coupling Fault (32)
6. Two Diagonal Cells (3)
7. Row Stuck-At (3)
8. Two Rows Stuck-At (3)
9. Partial Row Stuck-At (3)
10. Row Stuck-At 010101(2)
11. Column Stuck-At (3)
12. Two Columns Stuck-At (6)
13. Column Partially Stuck-At (3)
14. Row Stuck-At, Column Stuck-At (6)
15. Row, Partial Column Stuck-At (4)
16. Partial Row, Partial Column Stuck-At (4)
17. Two Columns, Two Rows Stuck-At (12)

define a row to be stuck-at 0. Thus, bitmaps generated for this fail pattern will all contain a row of cells all stuck at zero, but the row number is randomly chosen.

Given a bitmap, the generation of the fail matrix depends on the test input vector, which is determined by the test algorithm and the address scheme used. In this experiment, three popular production test algorithms, March C-, March C+, and Topological Checkerboard, are used. For each test algorithm, the test execution is simulated twice, once using the *fast column* and once using the *fast row* addressing scheme. Thus, for each bitmap generated, 6 (3 test algorithms x 2 address schemes) fail matrices are generated.

The experiment is repeated using three different memory specifications as listed in Table 2. The block heights in the first column are the same as the heights of the arrays. Multiplying the next two columns of the table, block widths and word widths, results in the widths of the memory arrays. The last column of the table, compression ratio, is derived by dividing the word widths, by the width of the compressor output, six. The compression ratio increases with the word width.

Table 2. Memory specification of software simulation

RAM	Block Height	Block Width	Word-Width	Compression Ratio
1	16	4	8	1.3
2	32	4	16	2.7
3	64	4	32	5.3

4.2 Simulation Results

In this section, the quality of this methodology will be measured by using the ratio of fully-reconstructed fail matrices to all fail matrices that were attempted, namely the *full reconstruction ratio.* Note that most partially-reconstructed fail matrices contain only very few *x*s and are still useful for manufacturing feedback and repairing purposes. Work is currently under progress to find an effective method to analyze partially reconstructed fail matrices.

The average full reconstruction ratios of different test algorithms and memory specifications are shown in Table 3. Notice that the ratios are consistently high, regardless of the test sequences or the address scheme used. In addition, this methodology performs equally well for different memory specifications, which determine the compression ratios.

Table 3. Average full reconstruction ratio of different test algorithms and addressing order

RAM	March C-		March C+		Top. Chkrbrd.	
	fast col	fast row	fast col	fast row	fast col	fast row
1	99.2%	100%	99.0%	100%	99.6%	100%
2	99.2%	100%	99.4%	100%	99.6%	100%
3	99.2%	100%	99.2%	100%	99.5%	100%

In Table 4, the average full reconstruction ratios of fail-pattern classes are shown. Full reconstruction occurs 100% of the time for fail pattern classes 1 to 15, and occurs most of the times for classes 16 and 17.

5 Hardware Implementation Experiment

The above method, along with two embedded SRAMs and a BIST circuit, is implemented in a test chip along with other test structures. A total of 12 wafers were manufactured in *a 0.18*µm mixed-signal process.

5.1 Design Specifications

The two memories used in this experiment are generated using a commercial memory compiler according to the following specifications:

Table 5. Memory specification of hardware implementation

RAM	Block Height	Block Width	Word Width	Compression Ratio
1	1024	16	32	5.3
2	128	8	23	3.8

The BIST circuit netlist was generated using a commercial BIST tool. The two memories are tested in series. The BIST executes three test algorithms to each memory in the

Table 4. Averaged full reconstruction ratio for each fail pattern class

Fail Pattern Class	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Full-Rectr. Ratio (%)	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	98.1	96.0

following order: March C- (fast row), March C+ (fast column), and Topological Checkerboard (fast column).

To accommodate two memories with different word widths, the compressor is designed to be reconfigurable for 32-bit and 23-bit input fail vectors. The BIST circuit signals the compressor whether the fail vector contains 23 bits or 32 bits.

The compression hardware consumes a total of 1581 equivalent NAND gates, and the BIST circuit consumes 1141 equivalent NAND gates. The compression hardware, designed in combinational circuit, computes only when the BIST circuit generates a fail vector, which does not happen every clock cycle. The compression hardware remains idle in the other clock cycles when the BIST circuit is not generating a fail vector. Thus, the hardware overhead may be reduced by being made sequential and utilizing originally idle clock cycles. However, the given time-constraint did not allow for the exploration of design options such as this one.

5.2 Implementation Results

As of the writing of this initial submission, the implemented circuits have been manufactured and are being tested. First results indicate that the proposed BIST methodology works in real life modern CMOS circuits. The complete results will be reported in the presentation at the conference.

6 Conclusion and Future Work

This paper presents a novel methodology to enable embedded memory diagnosis in pin-limited designs. The proposed method compresses the test responses and thus reduces the I/O pins needed. The test responses can later be reconstructed to compile bitmaps for diagnostic purposes.

Simulation with a variety of memory specifications, test algorithms and addressing schemes shows that this technique can be used for production testing. Simulation also shows that this technique works consistently well with a large variety of fail patterns. This method was implemented in silicon to assess its capabilities in real circuit.

Currently, a more efficient implementation to reduce the hardware overhead is being explored. In addition, work is underway to study the trade-off between compression quality and hardware overhead. For example, if a partially reconstructed matrix can be used for diagnosis, a high full reconstruction ratio will not be necessary, which would lead to “lighter” compression methodology and hardware.

Acknowledgments

The authors would like to thank Ian Burgess, Hans Heineken, Manuel D’Abreu, Saghir Shaikh, Laszlo Gutai, and Kenneth Walker, among others, for their support in this project.

References

[1] W. Maly and S. Naik, “Process Monitoring Oriented Testing,” *Proc. Int’l Test Conf.*, 1989, pp. 527-532.

[2] S. Naik, F. Agricola, and W. Maly, “Failure Analysis of High-Density CMOS SRAMs Using Realistic Defect Modeling and Iddq Testing,” *IEEE Design & Test of Computers*, Jun. 1993, pp. 13-23.

[3] A. L. Crouch, *DFT for Digital IC’s and Embedded Core Systems*, Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[4] C. Pyron et. al., “DFT Advances in Motorola’s MPC7400, a PowerPC Microprocessor,” *Proc. Int’l Test Conf.*, 1999, pp. 137-146.

[5] I. Schanstra et al, “Semiconductor Manufacturing Process Monitoring using Built-In Self-Test for Embedded Memories,” *Proc. of Int’l Test Conf.*, 1998, pp. 872-881.

[6] J. Khare et al, “SRAM-based Extraction of Defect Characteristics,” *Proc. of Int’l Conf. on Microelectronic Test Structures*, 1994, pp. 98-107.

[7] A.J. Van De Goor, *Testing Semiconductor Memories, Theory and Practice*, John Wiley & Sons, Chichester, UK, 1991.

[8] J. T. Chen and J. Rajski, “A Method and Apparatus for Diagnosing Memory using Self-Testing Circuits,” US Patent Pending, Appl. No. 09/522279.