A Built-In Self-Test Algorithm for Row/Column Pattern Sensitive Faults in RAM's

MANOJ FRANKLIN, KEWAL K. SALUJA, SENIOR MEMBER, IEEE, AND KOZO KINOSHITA, SENIOR MEMBER, IEEE

***Abstract* — Row and column sensitive faults in RAM's are a class of faults in which the contents of a cell become sensitive to the contents of the row and column containing the cell in presence of a fault. In this paper we formally define a fault model to include such faults and present an algorithm to detect faults from this fault model. We then describe two different implementations of the algorithm for a VLSI built-in-self-test (BIST) environment. They are: 1) a random-logic-based design, and 2) a microcode-based design. Finally we identify and list additional properties of the algorithm, such as its capability to detect stuck-at faults, coupling faults, and conventional pattern sensitive faults.**

I. Introduction

The monotonic increase in the density of random access memories (RAM's) necessitates a consistent effort to develop efficient test algorithms for large RAM's. It also brings forth the need to develop new fault models which can take into account the new types of physical failures that may arise due to high density, and are not modeled by the existing fault models. Existing test algorithms, developed either in an *ad hoc* manner [1] or on strong theoretical foundations [2]–[11], can detect faults from one or more of the following fault classes: stuck-type faults [2]–[6], pattern sensitive faults with five and nine cell neighborhoods [7]–[9], and coupling faults [5]. Although not all types of physical failures can be modeled by these fault classes, very little work has been done to investigate other classes of faults.

An important aspect to be considered while developing a new fault model is that the model should be sound and viable. One of the reasons for the lack of models covering complex fault classes is the belief that long tests would be required to detect such faults. In fact, the only tests that have been considered to be practical for large RAM's are those having a linear relationship with the number of bits N in the RAM. This constraint may be justifiable in a conventional testing environment where testing is done by expensive testers, and therefore test duration is of considerable importance (see Table I). In a built-in self-test (BIST) environment, all stages of testing can be carried out by relatively simple and inexpensive testers, requiring capabilities no more than powering up a chip, reading its status, etc.; much longer tests are therefore acceptable, provided they help in detecting a very large class of physical failures. It is evident from Table I that for very large RAM's, although $O(N^2)$ length tests may be unacceptable, $O(N^{3/2})$ length tests can be practical for memories of size 4M or even more, depending on the extent to which its internal organization can be exploited by the BIST logic [12]–[14]. Although BIST may still be a high-overhead concept for general integrated circuit designs, it requires little overhead for application in large RAM's, as will be shown in this paper. This fact has also been recognized in recent designs of large RAM's [18].

TABLE I
TEST TIME VERSUS TEST LENGTH, WITH A 10-MHz CLOCK

Order	Time	
	1M	4M
N	0.1 sec	0.4 sec
$N\log N$	2.1 sec	9.2 sec
$N^{3/2}$	1.8 min	14.3 min
N^2	30.5 hr	20.3 days

In our earlier work [15], [16], we proposed a *row/column weight sensitive fault model*, a model which encompasses most of the fault models studied in literature. In this fault model, the contents of a cell are assumed to be sensitive to the contents of the cells in its row and column, in the presence of a fault. Tests to detect faults from this model can be of an order no less than $O(N^{3/2})$ [15]. This necessitates the use of BIST to implement these tests. BIST logic can be realized either using random logic or microcode. Although random logic may offer the benefit of higher speed, the benefits of flexibility and ease of implementation offered by microcode offset the speed advantage of random logic. It can be argued that in the case of large RAM's, the testing speed is determined by the access time

Manuscript received July 28, 1989; revised October 30, 1989. This work was supported by the University of Wisconsin Graduate Research Committee and the National Science Foundation under Contract MIP 8509194.
M. Franklin is with the Computer Sciences Department, University of Wisconsin, Madison, WI 53706.
K. K. Saluja is with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.
K. Kinoshita was with the Department of Information and Behavioral Sciences, Hiroshima University, Hiroshima, Japan. He is now with the Department of Applied Physics, Osaka University, Osaka, Japan.
IEEE Log Number 8933560.

0018-9200/90/0400-0514$01.00 ©1990 IEEE

of the RAM and not the speed of the microcode, and hence the speed advantage of random logic is of little consequence. In this paper, we present both designs, but the microcode-based design is discussed in detail.

In Section II we describe the basics of the row/column weight sensitive fault model for the sake of completeness of this paper. Section III presents a conceptual explanation and working of the test algorithm given in the Appendix. In Section IV we briefly describe the random logic implementation, along with its area, time, and pin overheads. Section V gives the details of the microcode-based BIST design. Finally, Section VI provides additional properties of the algorithm, such as its capability to detect stuck-at faults, conventional pattern sensitive faults, and coupling faults.

II. Fault Model

Generally, an $N \times k$-bit RAM is organized as k identical partitions. Each N-bit partition may itself be organized as l ($l \geqslant 1$) two-dimensional arrays of $n \times m$ cells, such that $N = lmn$. Cells and their contents in each of these arrays are independent of the cells in other arrays. By assuming that no interaction can take place between cells of different arrays, we need to model faults only within a two-dimensional array. Therefore, from a testing point of view, it is only necessary to completely test each of these arrays, as opposed to testing the RAM as a single unit. These arrays can be tested either sequentially, or if the memory organization permits, many arrays could be tested in parallel. In any case, each array must be tested independent of the other arrays. For the sake of simplicity of presentation, we shall consider an array to be a square array of dimensions $n \times n$. Although our algorithm is presented for the square organization of an array, it is equally applicable to arbitrary $n \times m$ arrays. In the following definitions we shall assume that an array is organized as an $n \times n$-bit array.

- **Cell address**: The cells in the array are denoted as (i, j), where i and j denote the row and column addresses, respectively, within the array.
- **Base cell**: A cell in the memory array which is under test at a given instance.
- **Even cell**: A base cell with *even* row and column addresses.
- **Conjugate cells of an even cell** (i_e, j_e): The three cells defined by

$$\{([i_e - 1] \bmod n, j_e), (i_i, [j_e - 1] \bmod n), ([i_e - 1] \bmod n, [j_e - 1] \bmod n)\}.$$

  The test procedure explicitly tests only the even cells; when an even cell is tested, its conjugate cells are also tested.
- **Row neighborhood of a cell:** The row containing the cell, but excluding the cell.

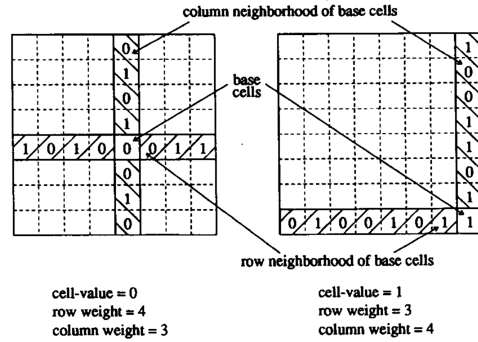


Fig. 1. Row/column neighborhood.

- **Column neighborhood of a cell**: The column containing the cell, but excluding the cell.
- **Cell value**: The content of the cell.
- **Row weight of a cell**: Number of 1's in its row neighborhood.
- **Column weight of a cell**: Number of 1's in its column neighborhood.

  Note that the row and column weights of a cell do not depend on its contents.
- **State of a cell**: The triplet (v, r, c), where v, r, and c are the cell value, row weight, and column weight, respectively, of the cell.

Each cell can have two possible cell values (0 or 1). The row and column weights of cells can vary between 0 and $n-1$. Therefore each cell can have $2n^2$ states. States in which the cell value of a cell is 0 (1) are called *states with cell value* 0 (1). Fig. 1 shows the row/column neighborhood, row weight, and column weight of two different cells.

Row and column pattern sensitive faults have been shown to exist in large RAM's using analytical arguments and simulation studies [11], [14]. You and Hayes have also observed [11] that the contents of a cell can be affected by the contents of cells in its row and column neighborhood. Interference could occur between cells which are in the same column or row [1], since these cells share common addressing and refresh circuitry. The neighborhood considered in this paper is essentially the "electrical neighborhood," and is much larger than the conventional five- and nine-cell neighborhoods.

Although ideally we may wish to consider sensitivity to arbitrary patterns of 0's and 1's in the row/column neighborhood, this will be impractical even for small n ($\geqslant 16$) because the number of patterns to be considered is $O(2^{2n})$. Therefore, we take a more pragmatic approach in defining the following fault model:

> **Row/column weight sensitive fault:** A memory cell is said to be faulty if it can never be in one or more of the $2n^2$ possible states defined by the triplet (v, r, c) for a cell.

In other words, a cell is said to be faulty if its content is sensitive to any combination of row and column weights.

For deriving the test algorithm for this fault model, we assume that READ and WRITE operations are fault-free, and that no more than a single fault is present in the memory array. Under the above assumptions, it can be shown that the least number of READ'S required to detect all multiple row/column weight sensitive faults is $2n^4$. If we consider only single faults, this bound reduces to $2n^2$. On the other hand, the minimum number of WRITE's required to test a memory for row/column weight sensitive faults (single or multiple) is $2n^4/2n-1 \approx n^3$. From the above statements, one can conclude that the minimum number of WRITE's, and hence the lower bound on the test length, for testing a memory for single row/column weight sensitive faults is $O(n^3) = O(N^{3/2})$ (assuming $N = n \times n$). Details of the justification of the model, the assumptions, their implications, and formal proofs of the above statements can be found in [15] and [16].

III. Conceptual Explanation and Working of Test Procedure

A simple method to test a memory for row/column weight sensitive faults is to test each cell independently. But this method requires at least $2N$ WRITE operations per cell, which amounts to a test length of at least $2N^2$. We argued in Section I of this paper that even in a BIST environment such tests may be impractical. Therefore it is imperative to find test sequences of length within $O(N^{3/2})$.

In the early stages of algorithm development, we made two observations which helped us in finding tests of length $O(N^{3/2})$. First, when a base cell is tested, its neighborhood cells also visit some of the $2N$ states. The test length can be substantially reduced by identifying these states and not repeating them when those cells are tested. Second, the test length can be kept close to the lower bound if we ensure that each WRITE operation contributes new states to as many cells as possible. With the insight gained from these observations, using simulation we developed and verified a number of algorithms with actual test lengths varying from $7n^3$ to $10n^3$. One such algorithm was reported in [15]. We found that, in general, the algorithms tend to become more complex as we try to decrease the test length, which precludes their implementation as BIST. For BIST application, we developed a simpler algorithm (of length $16n^3$), based on the observation that for every even cell in an array, there are three other cells (conjugate cells) that occupy symmetrical positions in the array. The algorithm can be made simpler (at the expense of increase in test length) by ensuring that the conjugate cells also get tested when an even cell is tested. One such algorithm, suitable for random logic implementation, was presented in [17] and implemented using random logic (Section IV). In this paper, we present another algorithm of low complexity (see Appendix) with a length of $16n^3$, suitable for microcode-based BIST design. An understanding of the working of this algorithm will be helpful in understanding the considerations involved in the BIST design presented in Section V.

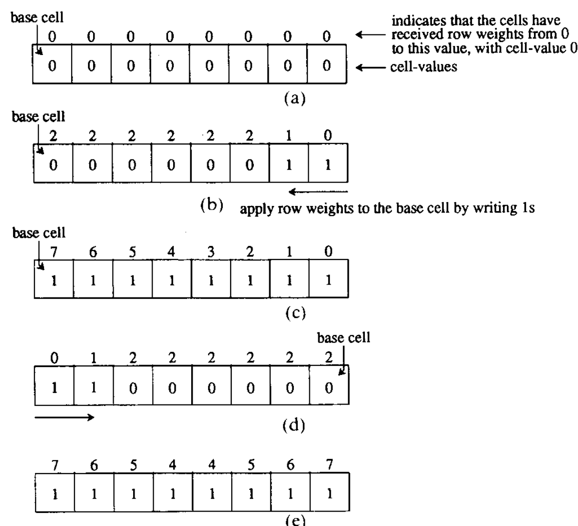


Fig. 2. Testing a linear array. (a) A linear array initialized to all 0's. (b) Application of tests to the leftmost cell as base cell. (c) State of the array after applying all row weights to the leftmost cell. (d) Application of tests to the rightmost cell as base cell. (e) Net effect of testing the leftmost and rightmost cells.

A. Conceptual Explanation

Let us consider a degenerate situation in which the memory array consists of a single row of eight cells. The row has been initialized to all 0's. Testing starts with the leftmost cell as the base cell (see Fig. 2(a)). At this stage, the cells have received only row weight 0, as indicated in Fig. 2(a). We apply row weights to the base cell by writing 1's from the rightmost cell onwards, as shown in Fig. 2(b). When all row weights from 0 to 7 have been applied to the base cell (Fig. 2(c)), its adjacent cell has received all row weights from 0 to 6, the next cell from 0 to 5, and so on; the rightmost cell (the conjugate cell of the base cell) has received only row weight 0, with cell value 0. But note that it has received all the row weights, with cell value 1.

To apply the states with cell value 0 to the rightmost cell, we repeat the WRITE operations, starting from the leftmost cell (Fig. 2(d)). In doing so, the leftmost cell gets states with cell value 1. The row weights received by the cells (with cell values 0) due to the combined effect of these two steps are shown in Fig. 2(e). Note that the middle cells have received the least number of row weights, as some of the states they received were repeated. However, when they become base cells, they need to be tested for fewer states. The remaining cells require even fewer number of states. Thus the test procedure explicitly applies different ranges of row weights to different cells. The same idea holds for the two-dimensional array.

The basic strategy adopted to test the memory is *divide and conquer* by recursive partitioning. The test procedure can be conceptually divided into the following steps:

1) *test the corner cells* of the array simultaneously, for states with cell values 0 and 1,
2) *test the border cells* for states with cell value 0,

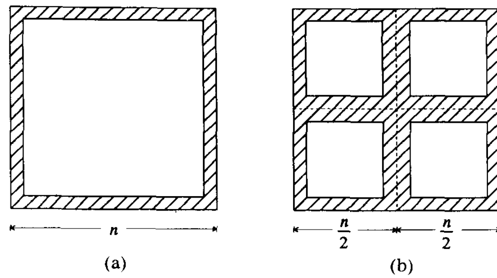


Fig. 3. Partitioning the memory array into four. (a) Memory array, with the border cells tested. (b) Memory array, partitioned into four partitions.

3) *test the interior cells* for states with cell value 0, by recursively partitioning the array, and
4) *repeat steps 2 and 3* for testing the non-corner cells for states with cell value 1.

Testing the corner cells for states with cell value 0 is explained in Section III-B-1. In the case of the corner cells, states with cell value 1 are also visited during the same procedure, as illustrated for the case of a linear array in this section.

For testing the border cells, let us consider the top row. We note that the middle cell $(0, n/2)$ has visited the minimum number of states. This cell is tested next. As with the corner cells, explicit testing of $(0, n/2)$ completely tests its conjugate cells, viz. $(0, n/2-1)$, $(n-1, n/2)$, and $(n-1, n/2-1)$. Testing of the middle cell $(0, n/2)$ and its conjugates divides the top row into two linear partitions, whose end cells have been tested completely. Each of these partitions is very similar to the original row with its end cells tested, and therefore can be further partitioned recursively, until the top row is completely tested. The bottom row cells also get tested simultaneously, they being the conjugates of the top row cells. The leftmost and rightmost columns are similarly tested.

After testing the border cells, the array is partitioned into four identical partitions by testing the two middle rows and columns. Each of the partitions obtained above is very similar to the original array with the border cells tested completely (Fig. 3), and therefore can be further partitioned in the same way as the original array was partitioned after its border cells were tested. We continue the process of partitioning the memory recursively in this manner. In each recursion step, we test *only* the two middle rows and the two middle columns of each partition until no more cells remain to be tested.

The above steps have so far tested all cells for states with cell value 0. Next, the array is initialized to all 0's, and steps 2 and 3 are repeated to test the non-corner cells for states with cell value 1.

B. Working of the Algorithm

The formal algorithm is given in the Appendix. The algorithm tests the array by explicitly testing the even cells using a procedure called *Procedure A*. For each even cell,

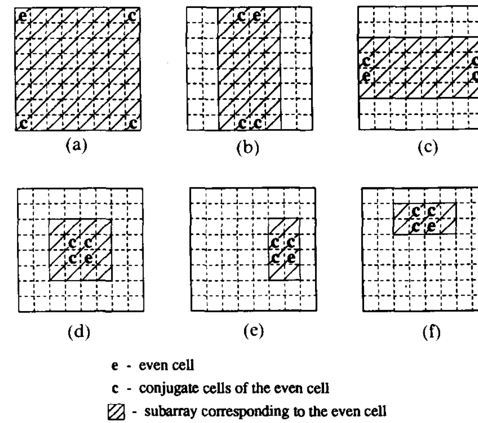


Fig. 4. Subarray and conjugate cells associated with different even cells: (a) (0,0); (b) (0,4); (c) (4,0); (d) (4,4); (e) (4,6); and (f) (2,4).

two parameters called p value and q value, and a subarray are defined based on its row and column address.

- **p value**: The p value, p_e, of an even cell (i_e, j_e) is the highest power of 2 less than or equal to n, such that $i_e \bmod p_e = 0$.
- **q value**: The q value, q_e, of an even cell (i_e, j_e) is the highest power of 2 less than or equal to n, such that $j_e \bmod q_e = 0$.

Example 1: For the cell (0,4) of an 8×8 memory array, the values of i, j, and n are 0, 4, and 8, respectively. The p and q values of this cell are 8 and 4, respectively. Similarly, for the cell (6,2), the i and j values are 6 and 2, and the p and q values are 2 and 2.

- **Subarray**: The subarray for the even cell (i_e, j_e) is a $p_e \times q_e$ array formed by the set of all cells $\{(i_s, j_s)\}$ such that

$$i_s = \left[i_e - \frac{p_e}{2} + \left(\frac{p_e}{2} + k\right) \bmod p_e\right] \bmod n, \qquad 0 \leqslant k \leqslant p_e - 1$$

$$j_s = \left[j_e - \frac{q_e}{2} + \left(\frac{q_e}{2} + k\right) \bmod q_e\right] \bmod n, \qquad 0 \leqslant k \leqslant q_e - 1.$$

Example 2: The subarray and conjugate cells associated with the cells (0,0), (0,4), (4,0), (4,4), (4,6), and (2,4) are shown in Fig. 4(a) to (f), respectively. Note that an even cell and its conjugate cells occupy symmetrical positions in the subarray.

When Procedure A tests an even cell, WRITE's are performed only on the cells within its subarray. This has the following two consequences: 1) cells within the subarray get tested to various extents, and 2) the subarray size regulates the extent of explicit testing done for each even cell. When an even cell is tested for states with cell value 0 (1), the portions of the array excluding the subarray are initialized and maintained at all 1's (0's). We shall explain

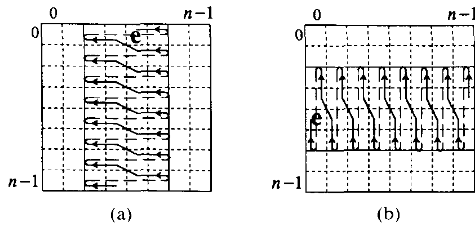


Fig. 5. Sequence of WRITE's performed by Procedure A while testing (a) an even cell e with $p \geq q$ and (b) an even cell e with $p < q$.

the working of Procedure A by showing how it tests a specific cell, viz. the corner cell (0,0), for states with cell value 0. The following term will be useful for the explanation.

Flip a cell is defined as reading the cell for the expected value and writing the complement of the expected value in the cell. It is important to note that the number of cells addressed during a flip operation is the same as the total number of arrays being tested in parallel. For example, in a 1-b-wide RAM organized as a single array, one cell at a time is addressed during reading and writing. In an 8-b-wide RAM with independent arrays, at least eight cells are addressed simultaneously. Now if each of these eight independent arrays is further organized as four arrays, and all these arrays can be tested simultaneously, then as many as 32 cells will be addressed simultaneously, during a flip operation.

1) Explanation of Procedure A (with respect to the cell (0,0)): When Procedure A begins, the array will be in the all 1's state. First, cells of the $n-1$th row are flipped; note that this amounts to reading a 1 and writing a 0 in every cell of row $n-1$, one cell at a time, involving n READ's and n WRITE's. At this stage, each cell in row $n-1$ will be in state $(0,0,n-1)$. Therefore, it is paramount to the application of at least row weight 0 and column weight $n-1$ with cell value 0, to every cell in row $n-1$. Next, row $n-2$ is flipped, then row $n-3$, and so on until row 0 is reached. This set of operations takes n^2 READ's and n^2 WRITE's. The succeeding operations are also flip operations and, therefore, a row/column fault excited in a cell by any of the patterns applied so far will be detected when the cell is read in its next flip operation.

Now the two conjugate cells $(n-1,0)$ and $(n-1,n-1)$ of (0,0) have visited all states with row weight 0 and cell value 0; also, the array state has become all 0's. For applying these states to (0,0) and the conjugate cell $(0, n-1)$, the above sequence is repeated, again starting from row $n-1$; note that now flipping amounts to reading 0's and writing 1's. The array state at this stage is all 1's; the total number of READ's and WRITE's performed so far is $4n^2$.

Next, for the application of row weight 1, the cells of column $n-1$ are flipped. Following this, the sequence mentioned in the previous two paragraphs is repeated. Similarly, for applying row weight 2, the same steps are repeated after flipping the cells of column $n-2$. This process continues until column 0 is reached. The array is then brought back to the all 1's state. Testing for each row weight takes $2n^2$ WRITE's, thus total of $2n^3$ WRITE's takes place in testing the four corner cells completely (the number of READ's is also $2n^3$). For an arbitrary even cell, the number of WRITE's is equal to $\min\{2pq^2, 2p^2q\}$.

2) Sequence of WRITE's: When Procedure A tests a cell, it applies all combinations of q number of row weights and p number of column weights. These combinations of weights can be applied to the cells in one of the two ways described below.

- Keeping the row weight fixed at each one of the values from $n-1-q$ to $n-1$, and varying the column weights from $n-1-p$ to $n-1$ or from $n-1$ to $n-1-p$, in steps of one.
- Keeping the column weight fixed at each one of the values from $n-1-p$ to $n-1$, and varying the row weights from $n-1-q$ to $n-1$ or from $n-1$ to $n-1-q$.

The order in which the subarray cells are read and written is different for both cases, and is shown in Fig. 5. For each cell, a method is chosen based on whether $p \geq q$ or $p < q$ so that the number of READ's and WRITE's is minimized.

IV. Random Logic Implementation

An algorithm suitable for random logic implementation was developed for BISTRAM and reported in [17]. This algorithm was implemented using the CAD tool Magic and a 2-μm CMOS technology with two metal layers, for a 4M memory [17]. In this section, we give a brief summary of the objectives and implementation aspects, as the general design methodology adopted for the microcode-based design presented in Section V is similar to this. The following objectives were set while completing the design and layout for the random logic implementation.

- Minimize the area occupied by the BIST hardware.
- Minimize the delay introduced to the normal memory operation.
- Minimize the number of additional pins required.

A. Organization

The BIST logic was divided into two parts: 1) control logic realized by finite state machines (FSM's), and 2) address generation logic. The FSM's were realized using random logic; the address generation logic was realized mostly using counters and multiplexers.

1) Finite state machines (control logic): The FSM's are the main control elements. They control the sequencing and start/stop of testing, and are hierarchically designed to map closely to the control flow in the algorithm.

2) Address generation logic: This unit consists of six 11-b ($\log_2 n$-bit) counters, which can be preset or cleared by the FSM's. This logic is similar to the address generation logic for the microcode-based design, the details of which are given in Section V-A-3.

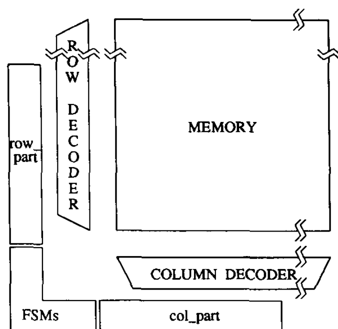


Fig. 6. Chip floor plan.

TABLE II
DEVICE COUNT AND SILICON AREA OF BIST LOGIC

Main parts	Device count	Silicon area	
		2μm technology (measured)	0.9μm technology (estimated)
FSMs	1064	1.125 mm^2	0.228 mm^2
row_part	1350	1.379 mm^2	0.279 mm^2
col_part	1704	1.510 mm^2	0.306 mm^2
Total	4118	4.014 mm^2	0.813 mm^2

B. Chip Floor Plan

The two parts of the BIST logic specified in Section IV-A are laid out as three separate regions. These are:

1) FSM's: layout of FSM 1 to FSM 6;
2) row_part: layout of the address generation logic for the row address;
3) col_part: layout of the address generation logic for the column address.

In the chip, these three regions are arranged in an L-shaped manner as shown in Fig. 6, to minimize global routing.

C. Overheads

The device count and the silicon area occupied by the different components of the BIST logic, for a 2-μm technology, are given in Table II. The total area of the BIST logic for a 2-μm technology is approximately 4 mm^2; note that the area can be substantially reduced by using aggressive design rules typically used in commercial memory designs. The area overhead, assuming a 0.9-μm technology and a nonaggressive design, would be around 0.8 mm^2. Considering the fact that a 4M CMOS DRAM fabricated using a 0.9-μm triple-poly, single-metal process has a chip area of about 100 mm^2 [19], the area occupied by the BIST logic will be no more than 0.8% of the chip area. If one considers area overhead based on device count, the overhead for a 4M CMOS DRAM is under 0.1%. Note that these estimates are rather pessimistic, and the actual overhead is likely to be even less.

The BIST logic needs only two additional pins—one for the clock and the other for initiating the testing. The fault signal can be multiplexed with a normal output pin if the performance penalty is acceptable. If the clock is available on-chip, then only one additional pin is required.

The maximum delay introduced to the normal inputs and outputs, for a 2-μm technology, is about 6 ns, as obtained from a timing simulator. This delay is due to the use of pass transistors in the multiplexer to do the selection between the normal address and the test logic generated address. This delay can be substantially reduced by using gates instead of pass transistors, and by using different design rules.

V. MICROCODE IMPLEMENTATION

The microcode-based BIST design was first proposed by Saluja and Kinoshita [20], [21]. They showed that for very large memories (4M or more), the area overhead for microcode-based design and random-logic-based design is identical. Microcode offers more flexibility of control than random logic, at the expense of speed and test time. In the case of BISTRAM, it may be even more area efficient than random logic since agressive design rules used for RAM cells can also be used for the microcode array. Therefore, the flexibility offered by microcode proves it to be more practical. Recently NEC used this approach in their 16-Mb BISTRAM [18]. In order to determine the amount of microcode and hardware required for a microcode-based implementation of our algorithm, we carried out a detailed logic design of the algorithm given in the Appendix. In this section, we present the details of this microcode-based design. In our design, the main control flow is implemented using microcode, and the support functions, such as address generation logic, are implemented using combinational logic and registers. This method offered a good trade-off because implementing the address generation functions using microcode would have required a large number of microinstructions. In completing the microcode-based design, we had the benefit of experience, as the random logic implementation preceded this design.

A. Organization

The BIST logic consists of three parts: 1) the microcode, 2) the decoder, and 3) the address generation logic. The microcode controls the sequencing; the decoder decodes the microinstructions to generate the signals required to control the control points. The address generation logic consists of random logic to implement the macros in the algorithm.

1) Microcode: Central to the BIST logic is the 15×10-b control store which controls the initialization, sequencing, and completion of testing. Though physically the control store is a single block of 15 words, logically it can be divided into four *microroutines*. These microroutines are analogous to the FSM's used for the random logic implementation, and they form a hierarchical stucture, as shown

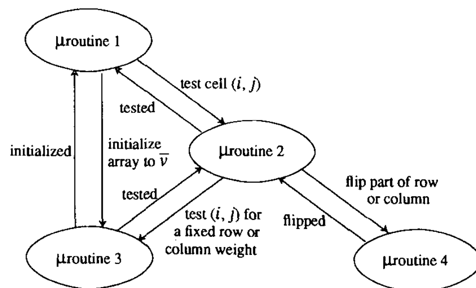


Fig. 7. Control flow among the microroutines.

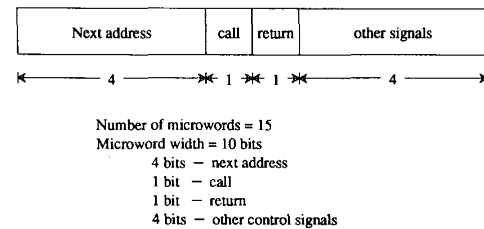


Fig. 8. Microinstruction format.

in Fig. 7. Control passes from one microroutine to another when the former issues a *call* to the latter; after the latter completes execution, control passes back to the former by way of a *return*. A 4-b *microstack*, of depth 2, is provided to store the return addresses, which helps in handling nested calls to the microroutines.

Microroutine 1: This microroutine supervises the testing of the whole array, by first performing the initialization (by calling microroutine 3), and then testing each even cell by updating the i and j registers (after the testing of each even cell is over), and calling microroutine 2 to test the cell (i, j). It consists of six microwords.

Microroutine 2: This implements Procedure A, and is responsible for finally carrying out the testing of each even cell. It sets up the *weight* register to the required value and iterates through a loop $p-1$ (or $q-1$) times, each iteration calling microroutine 3 twice and microroutine 4 once. It consists of four microwords.

Microroutine 3: This implements steps 2 and 4 of Procedure A. When called by microroutine 2, it tests the even cell for a fixed value (as decided by the *weight* register) of row weight or column weight. It is also used by microroutine 1 to initialize the memory array. When it is used for initialization purposes, the error detection capability of the BIST logic is disabled. Microroutine 3 consists of three microwords.

Microroutine 4: This implements step 5 of Procedure A and is used by microroutine 3 to flip some or all cells of a single row or column. It consists of two microwords.

Microinstruction Format and Encoding: The BIST logic has more than 20 control points, and if a microinstruction bit is provided for each control signal, it amounts to very wide microinstructions. Most of these control signals are not needed simultaneously. Furthermore, many signals are mutually exclusive. For example, only one register is decremented at any instant. For this reason, the signals were grouped and encoded, as shown in Fig. 8. Each microinstruction consists of 10 bits. A 4-b field is used for storing the *next address* for call and branch instructions. The next two bits are used for the *call* and *return* signals, respectively. The remaining four bits in conjunction with the two most significant bits of the μPC provide all the control signals. The decoding of these four bits is explained below.

2) Decoder: The decoder decodes the rightmost four bits of the microword to produce the necessary control signals required at the control points. The decoding is a two-step process. First, the two most significant bits of the μPC are decoded to determine how the four bits of the current microword have been encoded. This information is then used to decode the four bits. Though the decoding is done in two steps, this does not incur additional delay, because the first decoding step can be performed while waiting for the microinstruction fetch to complete.

3) Address Generation Logic: The address generation logic consists of five $\log_2 n$-bit registers: i, j, *free_i*, *free_j*, and *weight* registers. The i and j registers contain the row and column addresses of the even cell. The *weight* register is used to keep track of either the row weight or the column weight for which the even cell is currently being tested. The use of *free_i* and *free_j* registers is explained later. All five registers can be preset or decremented under the control of the microcode. The address generation logic also consists of combinational logic to implement the three macros in the algorithm, as explained below.

Macro CALCULATEpq: The function of this macro is to generate the p and q values. Combinational logic circuits called $p-1$ generator and $q-1$ generator, which take inputs from the i and j registers, respectively, are used to implement this macro. The logic to implement these functions is rather simple, as p and q are powers of 2.

Macro CHECKp.GE.q: In Section III-B-2, it was pointed out that two different sequences of WRITE operations are possible for testing a cell, and that for each cell, a particular sequence should be selected depending on whether $p \geqslant q$ or $p < q$. The function of macro CHECKp.GE.q is to implement the correct sequence by selecting the values for $PARAMETER1$, $PARAMETER2$, $FREE1$, and $FREE2$ of the algorithm. The macro is implemented in two parts. The first part is a combinational logic circuit which compares the p and q values produced by the $p-1$ generator and the $q-1$ generator, and generates the signal $p \geqslant q$. This logic is much simpler than a normal comparator because p and q are powers of 2. The second part is implemented within the decoder, which uses the signal $p \geqslant q$ to generate the appropriate control signals.

Macro CELL_ADDRESS: The function of this macro is to generate the cell address for the READ and WRITE operations. This also consists of two parts: row address generation and column address generation. These two parts are very similar, and we discuss only row address genera-

TABLE III
SEQUENCE OR ROW ADDRESSES GENERATED FOR READ AND WRITE OPERATIONS WHILE TESTING A CELL WITH $i = 01101011000$

free_i			Row address										
			$i-p$							0 or p	*free_i*		
1	1	1	0	1	1	0	1	0	1	0	1	1	1
1	1	0	0	1	1	0	1	0	1	0	1	1	0
1	0	1	0	1	1	0	1	0	1	0	1	0	1
1	0	0	0	1	1	0	1	0	1	0	1	0	0
0	1	1	0	1	1	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	0	1	1	0	1	0
0	0	1	0	1	1	0	1	0	1	1	0	0	1
0	0	0	0	1	1	0	1	0	1	1	0	0	0

tion. The expression for generating the row address is as follows (see Appendix):

$$\left[i - \frac{p}{2} + \left(\frac{p}{2} + free_i\right) \bmod p\right] \bmod n,$$

where $p-1 \geqslant free_i \geqslant 0$.

The terms in this expression can be rearranged to obtain

$$[(i-p)+0+free_i] \bmod n, \quad p-1 \geqslant free_i \geqslant (p/2)$$
$$[(i-p)+p+free_i] \bmod n, \quad (p/2)-1 \geqslant free_i \geqslant 0. \qquad (1)$$

Note that the variables of this expression are controlled by the microcode. The macro calculates the cell address, given the values of i and $free_i$. We shall illustrate the terms of the expression in (1) with the help of an example using binary values for i and p.

Example 3: Consider an even cell whose row address i is 01101011000. The p value for this cell is 1000. Therefore $free_i$ varies from 111 to 000. The value of the first term $i-p$ is 01101010000. For testing this cell, READ and WRITE operations have to be performed within its subarray. The sequence of row addresses, generated in accordance with the different values of the variable $free_i$, is given in Table III. Each entry in the table gives a value of $free_i$ and the corresponding row address split into three parts, as per the three terms of (1), namely $i-p$, 0 or p, and $free_i$.

Although expressions in (1) are the sum of three terms, the row address can be generated by concatenating appropriate bits of registers. This is also evident from Table III. The bits that need to be concatenated, along with the necessary logic blocks, are shown in Fig. 9. Only the least significant $\log_2 p$ bits of the $free_i$ register are used, since the maximum value of variable $free_i$ is $p-1$. The middle terms of the expressions in (1) can be implemented by using a 0 or 1 for the $\log_2 p$th bit, since p is a power of 2. The most significant $\log_2 n - (\log_2 p + 1)$ bits of the row address are obtained from the i register. Thus the row address is generated using multiplexers, as shown in Fig. 9.

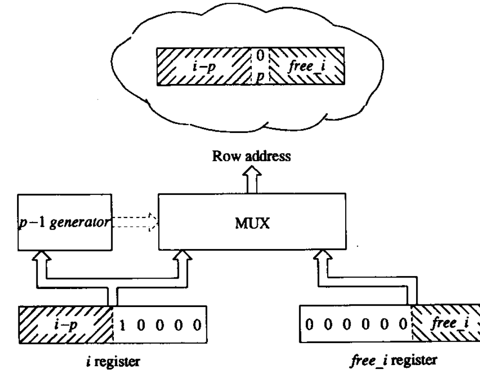


Fig. 9. Implementation of the row address generation part of the macro CELL_ADDRESS.

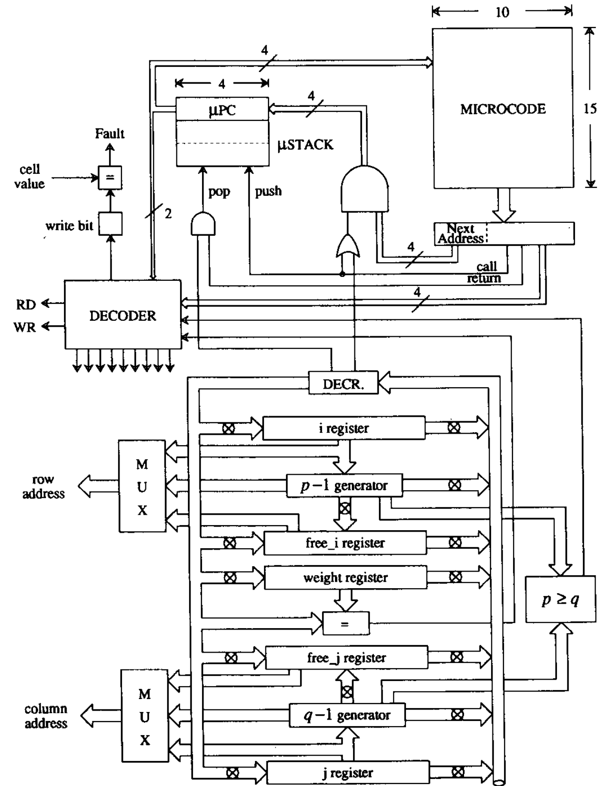


Fig. 10. Block diagram of the microcode-based BIST logic.

B. Discussion on Overheads

The complete block diagram of the microcode-based BIST logic is given in Fig. 10. The logic design of this unit was completed, but the layout was not performed for the following reasons. 1) The microcode-based design was done after the random logic implementation. The address generation logic for this design is simpler and more area efficient, because the counters used for the random logic design are replaced by registers and a shared decrement unit. 2) The algorithm proposed in this paper can be

TABLE IV
SUMMARY OF FAULTS DETECTED BY THE ALGORITHM

Test Procedure	Complexity	Detected Faults			
		Stuck-at-faults	Coupling faults	Restricted PSF	Row/Column
MSCAN	N	Does not cover decoder faults			
Marching 1 & 0	N	All	Does not cover all single coupling faults		
Checkerboard	N	All	Single coupling faults		
Suk and Reddy's Test A	N	All, if no coupling faults are present	All, if stuck-at-faults are not present		
Suk and Reddy's test procedure	N	All		Multiple PSF but not caused by read or non-transition writes	
Hayes' test procedure	N	All		Single PSF	
Shifted Diagonal	$N^{3/2}$	All	Single coupling faults		
Walk	$N^{3/2}$	All	Single coupling faults		
Row/Column	$N^{3/2}$	All	All unidirectional coupling faults	All unidirectional SPSF	Single and all unidirectional faults

implemented using a control store of size 15×10 b. A commercial DRAM design [18] has implemented a Marching test as BIST, using a control store of size 18×10 b. Thus, the area overhead for this algorithm is likely to be similar to the design given in [18]. 3) It can be argued that the time penalty in the memory access time for our design can be no worse than that of the designs given in [18] and [19]. This is because the only delay introduced to the normal inputs is due to the multiplexing of the normal addresses and the test logic generated addresses, which cannot be obviated in any BIST design. As far as the speed of testing is concerned, the main delay is the access time of the memory itself, because the control store is very small (15×10 b). The advantages of our algorithm by way of its increased fault detection capability are described in the next section.

VI. Additional Faults Detected

The test algorithm given in the Appendix has the additional capability of detecting all five cell neighborhood static pattern sensitive faults (SPSF's) in the memory array. In the SPSF model, a base cell is said to be faulty if its contents change from 0 to 1 and/or from 1 to 0 when a certain pattern of 0's and 1's exists in the neighboring cells [9]. When a memory array is tested by our algorithm, each cell receives the 32 different five-cell neighborhood patterns, with some of the patterns occurring many times. The details of how and when a base cell receives all the patterns can be found in [17].

The test algorithm given in this paper also covers most of the *ad-hoc* test algorithms developed earlier. Table IV provides a list of those test procedures which form a subset of our algorithm. Blank entries in Table IV indicate the class of faults that are either not detected or covered only to a limited extent.

The Marching test and a scan WRITE/READ test with a checkerboard pattern have been implemented using a microcode by NEC in their recent 16-Mb RAM chip [18]. The control store size for their design is 18×10 b, and the area overhead is less than 1%. Though these two algorithms have been implemented, the fault coverage offered by those algorithms is rather poor. Implementation of our algorithm requires an even smaller control store of size 15×10 b. Considering the fact that our algorithm detects most of the conventional faults in addition to the row/column weight sensitive faults, implementation of the algorithm in RAM chips will definitely be an advantage.

VII. Conclusions

A new fault model has been developed for random access memories for a class of pattern sensitive faults called row/column weight sensitive faults. A test procedure has been developed to detect faults from the developed fault model. This test procedure also detects five-cell neighborhood static pattern sensitive faults and other conventional faults such as stuck-at faults and coupling faults. Two different implementations of the algorithm for a VLSI built-in self-test environment were discussed. The first design, using random logic, was implemented by completing the logic design and layout in 2-μm CMOS technology. The silicon area overhead for a 4M RAM is as little as 0.8%. The details of the second design, using microcode, are also presented. This design can be implemented with a microcode as small as 15×10 b.

APPENDIX

The formal test procedure is presented below in a Pascal-C-like notation.

Algorithm

1. Initialize the array to 1;
2. Call Procedure A (0,0,0);
3. **For each** even cell (i, j) except (0,0) **do**
   Call Procedure A $(i, j, 0)$;
4. Initialize the array to 0;
5. **For each** even cell (i, j) except (0,0) **do**
   Call Procedure A $(i, j, 1)$;

Procedure A (i, j, v) /* for cell (i, j), for states with cell value v */

1. CALCULATEpq;
   CHECKp.GE.q;
   $weight := PARAMETER2 - 1$;
2. **for** $FREE1 := PARAMETER2 - 1$ to 0 **do**
   for $FREE2 := PARAMETER2 - 1$ to 0 **do**
   CELL_ADDRESS;
   if $FREE2 \leqslant weight$ **then** read($\bar{v}$); write (v);
   else read (v); write ($\bar{v}$);
3. **if** $weight < 0$ **then stop**
4. Repeat step 2, with v in place of $\bar{v}$, and $\bar{v}$ in place of v.
5. $FREE2 := weight$;
   for $FREE1 := PARAMETER1 - 1$ to 0 **do**
   CELL_ADDRESS;
   read($\bar{v}$); write (v);
6. *weight*--;
   go to step 2.

Macro CALCULATEpq

$p = \min\{n, \max\{2^x\}\}; \quad i \bmod 2^x = 0;$

$q = \min\{n, \max\{2^x\}\}; \quad j \bmod 2^x = 0;$

Macro CELL_ADDRESS

$$\text{row} := \left[i - \frac{p}{2} + \left(\frac{p}{2} + free_i \right) \bmod p \right] \bmod n;$$

$$\text{column} := \left[j - \frac{q}{2} + \left(\frac{q}{2} + free_j \right) \bmod q \right] \bmod n;$$

Macro CHECKp.GE.q

If $p \geqslant q$ **then**	**else**
$PARAMETER1 = p$;	$PARAMETER1 = q$;
$PARAMETER2 = q$;	$PARAMETER2 = p$;
$FREE1 = free_i$;	$FREE1 = free_j$;
$FREE2 = free_j$;	$FREE2 = free_i$;

ACKNOWLEDGMENT

The authors would like to thank the referees for their comments and suggestions that greatly enhanced the quality of the presentation of this work.

REFERENCES

[1] M. M. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Potomac, MD: Computer Science Press, 1976.

[2] W. Barraclaugh, A. C. L. Chiang, and W. Sohl, "Techniques for testing the microcomputer family," *Proc. IEEE*, vol. 64, no. 6, pp. 943–950, June 1976.

[3] J. Knaizuk, Jr. and C R. P. Hartmann, "An optimal algorithm for testing stuck-at-faults in random access memories," *IEEE Trans. Computers*, vol. C-26, no. 11, pp. 1141–1144, Nov. 1977.

[4] S. M Thatte and J. A. Abraham, "Testing of semiconductor random access memories," *Proc. FTCS-7*, pp. 81–87, June 1977.

[5] R. Nair, S. M. Thatte, and J. A. Abraham, "Efficient algorithms for testing semiconductor random access memories," *IEEE Trans. Computers*,, vol. C-27, no. 6, pp. 572–576, June 1978.

[6] R. Nair, "Comments on an optimal algorithm for testing stuck-at faults in random access memories," *IEEE Trans. Computers*, vol. C-28, no. 3, pp. 258–261, Mar. 1979.

[7] J. P. Hayes, "Detection of pattern sensitive faults in random access memories," *IEEE Trans. Computers*, vol. C-24, no. 2, pp. 150–157, Feb. 1975.

[8] D. S. Suk and S. M. Reddy, "Test procedures for a class of pattern sensitive faults in random access memories," *IEEE Trans. Computers*, vol. C-29, no. 3, pp. 419–429, June 1980.

[9] K. K. Saluja and K. Kinoshita, "Test pattern generation for API faults in RAM," *IEEE Trans. Computers*, vol. C-34, no. 3, pp. 284–287, Mar. 1985.

[10] M. S. Abadir and H. K. Reghbati, "Functional testing of semiconductor random access memories," *Computing Surveys*, vol. 15, no. 3, pp. 175–198, Sept. 1983.

[11] Y. You and J. P. Hayes, "A self-testing dynamic RAM chip," in *Proc. Conf. Advanced Res. VLSI*, Jan. 1984, pp. 159–168; also in *IEEE J. Solid-State Circuits*, vol. SC-20, no. 1, pp. 428–435, Feb. 1985.

[12] M. Malek, "Two dimensional multiple access testing technique of RAM," in *Proc. ICCD*, Oct. 1986, pp. 248–251.

[13] P. Mazumder and J. H. Patel, "A novel fault-tolerant design of testable dynamic random access memory," in *Proc. ICCD*, Oct. 1987, pp. 306–309.

[14] K. T. Le and K. K. Saluja, "A built-in self-testing design of random access memory (BISTRAM)," in *Proc. Int. Test Conf.*, Sept. 1986, pp. 830–839.

[15] M. Franklin, K. K. Saluja, and K. Kinoshita, "Row/column pattern sensitive fault detection in RAMs via built-in self-test," *Proc. FTCS-19*, pp. 36–43, June 1989.

[16] M. Franklin, K. K. Saluja, and K. Kinoshita, "An algorithm for the detection of row/column pattern sensitive faults in RAMs," Univ. of Wisconsin-Madison, Tech. Rep. ECE-89-5, 1989.

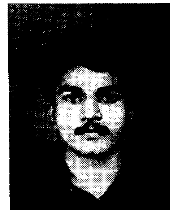
[17] M. Franklin, K. K. Saluja, and K. Kinoshita, "Design of a BISTRAM with row/column pattern sensitive fault detection capability," in *Proc. Int. Test Conf.*, Aug. 1989, pp. 327–336.

[18] T. Takeshima *et al.*, "A 55 ns 16 Mb DRAM," in *ISSCC Dig. Tech. Papers*, Feb. 1989, pp. 246–247, 353.

[19] T. Ohsawa *et al.*, "A 60-ns 4-Mbit CMOS DRAM with built-in self-test function," *IEEE J. Solid-State Circuits*, vol. SC-22, no. 5, pp. 663–668, Oct. 1987.

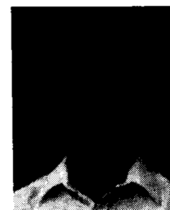
[20] K. Kinoshita and K. K. Saluja, "Built-in testing of memory using an on-chip compact testing scheme," *IEEE Trans Computers*, vol. C-35, no. 10, pp. 862–870, Oct. 1986.

[21] K. K. Saluja, S. H. Sng, and K. Kinoshita, "Built-in self-testing RAM: A practical alternative," *IEEE Design & Test Computers*, pp. 42–51, Feb. 1987.



Manoj Franklin was born in Trivandrum, India, in 1962. He received the B.Sc. degree in electronics and communications engineering from the University of Kerala, India, in 1984.

He then joined BHEL, Bangalore, India, where he worked for three years. He is currently working towards the Ph.D. degree in Computer Sciences at the University of Wisconsin, Madison. His main research interests are digital circuit testing, fault-tolerant computing, and computer architecture.



Kewal K. Saluja (S'70–M'73–SM'89) received the B.E. degree in electrical engineering from the University of Roorkee, Roorkee, India, in 1967, and the M.S. and Ph.D. degrees from the University of Iowa, Iowa City, in 1972 and 1973, respectively.

He is currently an Associate Professor in the Department of Electrical and Computer Engineering at the University of Wisconsin, Madison, where he teaches logic design, computer architecture, microprocessor-based systems, VLSI design and testing. Previously he was at the University of Newcastle, Australia.

He has also held visiting and consulting positions at a number of institutions such as the University of Southern California, the University of Iowa, and Hiroshima University. His research interests include design for testability, fault-tolerant computing, VLSI design, and computer architecture.

Kozo Kinoshita (S'58–M'64–SM'89) received the B.E., M.E., and Ph.D. degrees in engineering from Osaka University, Osaka, Japan, in 1959, 1961, and 1964, respectively.

From 1964 to 1966 he was an Assistant Professor and from 1967 to 1977 he was an Associate Professor of Electronic Engineering at Osaka University, Osaka, Japan. From 1978 to 1989 he was a Professor in the Department of Information and Behavioral Sciences, Hiroshima University, Hiroshmia, Japan. He is presently a Professor in the Department of Applied Physics, Osaka University, Osaka, Japan. His fields of interest are logic design, fault diagnosis, and fault-tolerant designs of digital systems.

Dr. Kinoshita is a member of the Institute of Electrical Engineers of Japan, the Institute of Electronics, Information and Communication Engineers of Japan, and the Information Processing Society of Japan.