Fault Simulation and Test Algorithm Generation for Random Access Memories

Chi-Feng Wu, *Student Member, IEEE*, Chih-Tsun Huang, *Member, IEEE*, Kuo-Liang Cheng, *Student Member, IEEE*, and Cheng-Wen Wu, *Senior Member, IEEE*

***Abstract*—The size and density of semiconductor memories is rapidly growing, making them increasingly harder to test. New fault models and test algorithms have been continuously proposed to cover defects and failures of modern memory chips and cores. However, software tool support for automating the memory test development procedure is still insufficient. For this purpose, we have developed a fault simulator (called RAMSES) and a test algorithm generator (called TAGS) for random-access memories (RAMs). In this paper, we present the algorithms and other details of RAMSES and TAGS and the experimental results of these tools on various memory architectures and configurations. We show that efficient test algorithms can be generated automatically for bit-oriented memories, word-oriented memories, and multiport memories, with 100% coverage of the given typical RAM faults.**

***Index Terms*—Fault simulation, integrated-circuit testing, memory testing, RAM, semiconductor memory, test pattern generation.**

I. Introduction

SEMICONDUCTOR memories are widely considered to be one of the most important types of microelectronic components in modern digital systems [1]. It is reported that memories represent about 30% of the worldwide semiconductor market [2]. The growing need for storage in computer, communications, consumer, and network applications is driving the continuous innovation of various semiconductor memory technologies. Bigger and faster memories are always desirable due to our insatiable appetites for voluminous transmission and storage of data in these applications, i.e., the continuing technology innovation is likely to increase the market share of commodity and embedded memories in the future. The increasing size and density of memory chips will soon make their testing the bottleneck of the entire production process. Yield loss is another issue due to the increased size and density. Memories are more vulnerable to physical defects than logic circuits because of their higher density and more complicated processing steps. Therefore, investing in memory failure analysis, fault modeling and simulation, test algorithm development and evaluation, DFT, built-in self-test (BIST), diagnostics, etc., has been considered one of the key factors in producing successful memory as well as system-on-chip (SOC) products. Tools for fault model evaluation and test algorithm generation are fundamental for tackling the above issues efficiently.

Manuscript received April 22, 2001; revised October 8, 2001 and December 15, 2001. This paper was recommended by Associate Editor R. Aitken.

The authors are with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu 30013 Taiwan R.O.C. (e-mail: cww@ee.nthu.edu.tw).

Publisher Item Identifier S 0278-0070(02)02470-3.

Functional fault models are commonly used for memories. They define the functional behavior of the faulty memory. More and more fault models are being proposed to cover defects and failures in modern memory circuit and deep-submicron process technologies. Test algorithms are also being developed to detect these faults. Many works on fault models and test algorithms have been reported in the past for random access memory (RAM), including SRAM and DRAM (see, e.g., [1]–[9]). It is known that the effectiveness of memory test algorithms can be obtained by manual analysis [5]. However, as more and more complicated memory architectures and fault models are being introduced, it is becoming evident that fault simulation is more feasible than manual analysis. Memory fault simulation is better than manual analysis in at least the following aspects [10]: 1) fault coverage evaluation can be done efficiently, especially when the number of fault models is large; 2) in addition to bit-oriented memories, word-oriented memories can be simulated easily even with multiple backgrounds; 3) test algorithm design and optimization can be done in a much easier way; 4) detection of a test algorithm on unexpected faults can be discovered; and 5) fault dictionary can be constructed for easy diagnosis. An earlier work on automatic functional fault coverage analysis was presented by Riedel and Rajski [7]. They proposed a fault coverage evaluation method based on fault state transition (FST). The basic idea of the FST method is to keep track of the sensitization and desensitization information for each fault by a table. The advantage of the method is its flexibility in adding new fault models. However, it has the drawback of a rapid-growing FST table size and table lookup time with respect to the number of fault models and the size of the memory under simulation. Word-oriented memories and multiport memories were not supported.

In this paper, we first present a fast memory fault simulator called the random access memory simulator for error screening (RAMSES), which consists of a simulation engine and numerous fault descriptors. It is a much improved version of that presented in [10]. The simulation engine reads the test inputs and sets the operation flags for each memory cell. Fault coverage is determined by checking the fault descriptors for predefined conditions. New fault models can be covered by adding their fault descriptors. Also, RAMSES supports bit- and word-oriented memories, as well as multiport memories.

The proposed fault simulator also can be used for test algorithm generation. March-based test algorithms have been considered the most efficient for stuck-at, transition, stuck-open, address decoder, and coupling faults. March tests are easy to generate and are normally short. Previous research efforts have been focused on finding short test algorithms targeting certain fault models [11]–[13]. The fault coverage of the test algorithms was proved by analysis using mathematical models such as state diagrams. Mathematical fault modeling and the finite-state machine (FSM) method can be used to generate the March test for full coverage of a certain set of faults, which can then be verified for completeness or irredundancy. Recently, a systematic approach has been proposed that converts tests for bit-oriented RAM to tests for word-oriented RAM with complete fault coverage [13]. The transition matrix model proposed in [14] provides a more detailed description of the fault models and test sequence optimization. Though elegant, these mathematical approaches have some restrictions. They do not provide fault coverage figures during the test generation and verification process or the data background selection steps. When more new and complex fault models are introduced, deriving a test algorithm becomes difficult. Test algorithm optimization is even harder, because the background selection for word-oriented memories and the port selection for multiport memories are quite complex and will vary for different memory architectures.

We propose in this paper a RAMSES-based memory test generation methodology that is flexible for different fault models and memory architectures. It is called test algorithm generation by simulation (TAGS). Most popular RAM fault models are supported and new fault models can be added easily thanks to the flexibility of the simulation and test generation algorithms. Multiple data backgrounds and multiport memories also are supported.

The remainder of the paper is organized as follows. Section II reviews the fault models and definitions that we use in this paper. Fault simulation issues are described in Section III and the test algorithm generator is presented in Section IV. Finally, we conclude the paper in Section V.

II. Fault Models and Definitions

We briefly review the RAM fault models that are considered in this paper, following the notation and definitions given in [5]. A *stuck-at fault* (SAF) occurs when the value of a cell or line is always 0 (a stuck-at-0 fault) or always 1 (a stuck-at-1 fault). A *stuck-open fault* (SOF) occurs when the cell cannot be accessed due to a broken word line. A read to this cell will produce the previously read value. An *address decoder fault* (AF) is a functional fault in the address decoder that results in faulty address accesses. A cell has a *transition fault* (TF) if it fails to transit from 0 to 1 (a $\langle\uparrow/0\rangle$ TF) or from 1 to 0 (a $\langle\downarrow/1\rangle$ TF). A *coupling fault* (CF) between two cells occurs when the logic value of a cell is influenced by the content of, or operation on, another cell. There are several types of coupling faults. An *inversion coupling fault* (CFin) occurs if a transition in one cell inverts the logic value of another. There is an *idempotent coupling fault* (CFid) if a transition in one cell forces a fixed logic value into another. There is a *state coupling fault* (CFst) if a cell or line is forced to a fixed logic value only if the coupling cell or line is in a given state. Finally, there is a read disturb fault (RDF) if the cell value will flip when being read [15].

TABLE I
Some March Algorithms

Name	Faults detected
Algorithm	
MATS++	AF/SAF/SOF
$\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0, r0)$	
March X	AF/SAF/TF/CFin
$\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0); \Updownarrow(r0)$	
March Y	AF/SAF/TF/SOF/CFin
$\Updownarrow(w0); \Uparrow(r0, w1, r1); \Downarrow(r1, w0, r0); \Updownarrow(r0)$	
March C−	SAF/AF/TF/CF
$\Updownarrow(w0); \Uparrow(r0, w1); \Uparrow(r1, w0); \Downarrow(r0, w1); \Downarrow(r1, w0); \Updownarrow(r0)$	

Many March-based test algorithms have been proposed to detect the above faults [5], [10], [16]. We list a few examples in Table I, using the notation as defined in [5] and [16].

For a *word-oriented memory*, we let N represent the number of data words in the memory, each word having w bits. In this case, the Read/Write operations in the March tests are extended to reading and writing a word (called the *background word*, *background pattern*, or *data background*) instead of a bit at a time. For example, the word-oriented MATS++ is represented as $\{\Updownarrow(wa); \Uparrow(ra, w\bar{a}); \Downarrow(r\bar{a}, wa, ra)\}$, where a is a background word [17]. Fault models listed above were originally developed for bit-oriented memories. Faults that occur on a single cell, e.g., SAF, can still be used for word-oriented memories. Faults involving two or more cells, however, should be further classified according to whether they are within the same word or not, i.e., intraword or interword faults [13]. Traditionally, standard data backgrounds are used to test a word-oriented memory for intraword coupling faults. However, by RAMSES we have developed a more efficient class of test algorithms, the *Cocktail-March* algorithms and have derived a shortest one (the *March-CW* algorithm) ever reported so far for covering SAF, AF, TF, SOF, CFst, CFid, and CFin [10].

Conventionally, a multiport RAM is tested similar to a single-port RAM under the same fault models, by applying the same test algorithm repeatedly to each port or each pair of ports. This approach is insufficient for detecting interport faults. A two-port memory array example is depicted in Fig. 1, where the ports are denoted as Port A and Port B. The interport short fault is likely to occur on adjacent word lines or adjacent bit lines. Dedicated fault models are necessary if the detection of interport shorts is desired [18], [19]. Some other interport faults have been investigated in [15], where a complete set of all possible interport faults are also defined. The set is very large and the corresponding test is quite long. In most cases only the shorts need to be considered. In this paper, we consider the interport shorts in addition to the single-port faults as mentioned above.

The interport shorts can be classified as the *bit-line short fault* (BSF) and *word-line short fault* (WSF) [20]. Fig. 1 shows an example, in which some defects result in a BSF and a WSF. The bit lines are drawn in a simplified way for ease of presentation, though we actually consider differential pairs and all possible shorts during fault simulation. When there is an interport WSF between Cells 1 and 3 as shown in Fig. 1, a possible result is as shown in Fig. 2, when we access Cell 1 through Port A and Cell 2

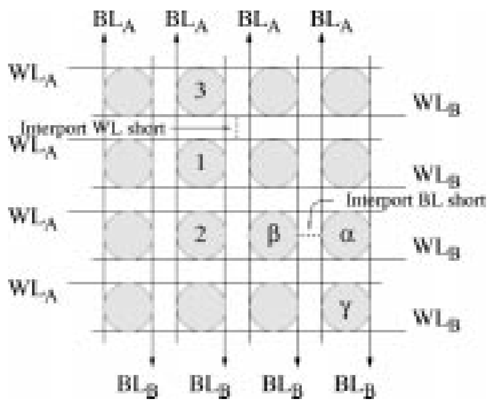


Fig. 1. A two-port memory array example.

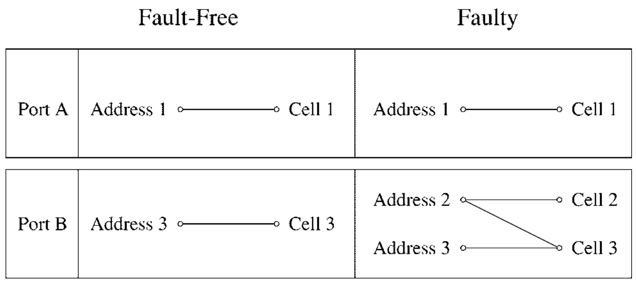


Fig. 2. Behavior of an interport WSF.

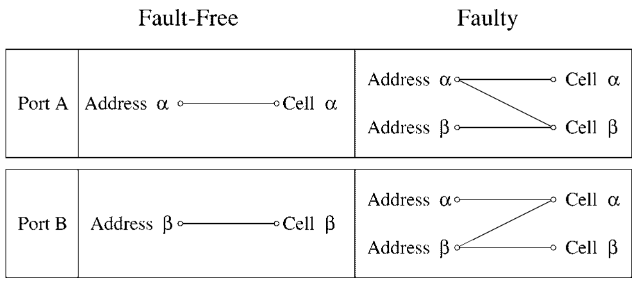


Fig. 3. Behavior of an interport BSF.

through Port B simultaneously. Due to the short fault, Port B has a multiple access to Cell 2 and Cell 3 when Port A is accessing Cell 1. The resulting value of a Read to multiple cells depends on the memory design: possible faulty results are the logic-AND or logic-OR of the two cells. Also, when the interport BSF (as in Fig. 1) occurs, a possible result is as shown in Fig. 3. The Port A address of Cell α can simultaneously access Cells α and β, so does the Port B address of Cell β. Again, the resulting value of a Read to multiple cells can be the logic-AND or logic-OR of the two cells, depending on the memory design. Note that an interport short can lead to multiple faults on the same bit line or word line, e.g., Cell 2 discussed above can be any cell in the column other than Cells 1 and 3.

III. Fault Simulation

Memory fault simulation is different from logic fault simulation in many ways. First of all, unlike a logic circuit, memory has a regular structure. It consists of one or more cell arrays and the peripheral read/write circuits. However, there are many memory architectures and configurations that come with various address sizes, word lengths, and port numbers. Secondly, in a logic circuit we usually use only stuck-at faults, but there are multiple fault models for memories as discussed above. Furthermore, instead of the single-fault models, we have to assume multiple faults—each of the fault models can appear multiple times in the memory circuit at different locations. Finally, parallel simulation techniques developed for logic fault simulation are not suitable for simulating memory faults. Therefore, it is necessary to develop fault simulation algorithms that are dedicated for memories.

```
for each $i$, $0 \le i \le k-1$, begin
    inject $f_i$ to $M$;
    for each $j$, $0 \le j \le s-1$, begin
        apply $t_j$ to $M$;
        if (output neq fault_free_output) begin
          set_detect($f_i$);
          break;
        end-if
    end-for
end-for
```

Fig. 4. The sequential fault simulation procedure.

In this section, we will introduce the sequential fault simulation first, which is a general and straightforward simulation algorithm. The space complexity and time complexity will be evaluated. After that, we will propose an improved fault simulation algorithm for implementing RAMSES, our fast memory fault simulator. Fault simulation techniques for word-oriented memories and multiport memories will also be presented.

A. Sequential Fault Simulation

In sequential fault simulation, faults are injected into the system one by one and test patterns are applied to each and every faulty system. Outputs are then observed for evaluating the fault coverage (FC) of the test patterns. Consider a memory M that consists of N bits of data, with the list of target faults $f_0, f_1, \ldots, f_{k-1}$. The sequence of test patterns is $t_0, t_1, \ldots, t_{s-1}$. The sequential fault simulation procedure is shown as Fig. 4.

The time complexity of the sequential fault simulation is $T = k \times s$, where k is the fault count and s is the length of the test algorithm (i.e., number of test patterns). For single cell faults, $k = O(N)$, but for two-cell coupling faults, $k = O(N^2)$. Also, for March tests, $s = O(N)$. Therefore, the time complexity for sequential fault simulation under March tests is $T = O(N^3)$, when two-cell coupling faults are the most complex faults in the target fault set. Furthermore, the space complexity of sequential fault simulation is dominated by the fault count, i.e., $O(N^2)$ in this case.

Due to the high complexity, sequential fault simulation is obviously not a practical solution for real-world applications. Nevertheless, it is easy to implement and is still useful for verifying the correctness of other simulation algorithms.

We have noted that it is feasible to simulate a smaller version of the memory under test for the purpose of FC evaluation, because of the regularity in memory structures [10]. Simulation results of a small memory (e.g., 1 Kb) are the same with a large

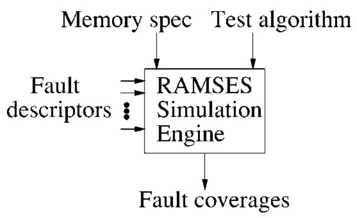


Fig. 5. RAMSES overview.

TABLE II
Two Fault Descriptors

RDF⟨$r0/\uparrow$⟩	CFin⟨$\uparrow;\updownarrow$⟩
`AGR := R0`	`AGR := UTR`
`SPT := @`	`SPT := *`
`VTM := R0`	`VTM := R0, R1`
`RCV := W0, W1`	`RCV := W0, W1`

one (e.g., 16 Mb) for most fault models, though scaling requires certain calculation to avoid FC error. We also have noted that the sequential simulation algorithm has a high percentage of redundancy. We will propose an improved algorithm next.

B. RAMSES

RAMSES is a fast memory fault simulator that features the novel *fault descriptor* concept. As illustrated in Fig. 5, the simulator consists of the RAMSES simulation engine and numerous fault descriptors. The simulation engine executes the RAMSES simulation algorithm, which is not dedicated to a specific fault model. Target fault models are defined by their specific fault descriptors. For a user-defined memory specification and a test algorithm, RAMSES reports the FC for each fault model that is defined by a fault descriptor.

A fault descriptor consists of four primary attributes.

1) The aggressor (AGR) is an operation or condition which can activate the fault effect.
2) The victim (VTM) is the operation affected by the fault, i.e., it will produce an observable faulty output.
3) The suspect (SPT) is the possible location(s) of the aggressor and it also indicates the possible victim location(s) for each aggressor.
4) The recoverer (RCV) is the operation which can mask or recover the fault effect on the victim.

Two fault descriptor examples are listed in Table II. For the read disturb fault, i.e., RDF⟨$r0/\uparrow$⟩, a Read-0 operation activates the fault in the local cell, so the aggressor is the Read-0 operation and the suspect is the local cell represented by "@." The victim is also a Read-0 operation. Since the faulty cell's value will be flipped to 1 after the Read operation, a Read-0 operation can detect the fault. However, if the cell is written a 0 or a 1 before the victim operation takes place, then the cell value is recovered and the fault effect is erased. Therefore, the recoverer can be a Write-0 operation or a Write-1 operation.

The second example shows that CFin⟨$\uparrow;\updownarrow$⟩ is activated by an up transition of the cell. It can be observed by either a Read-0 or Read-1 operation at the victim cell and can be recovered by either a Write-0 or Write-1 operation at the victim cell. The "*" means that all other cells are possible coupling cells and it also indicates that for any aggressor cell, all other cells are possible victim cells. The core algorithm for RAMSES is summarized as Fig. 6.

```
for each operation begin
    set_op_flags;
    if (AGR ⊂ op_flags) begin
      for each victim cell begin
          set victim flags;
          set aggressor address;
      end-for
    end-if
    if (OP eq RCV) begin
      clear victim flag;
      clear aggressor entry;
    else if (OP eq VTM) begin
      mark detected;
      end-if
    end-if
end-for
```

Fig. 6. The algorithm for RAMSES.

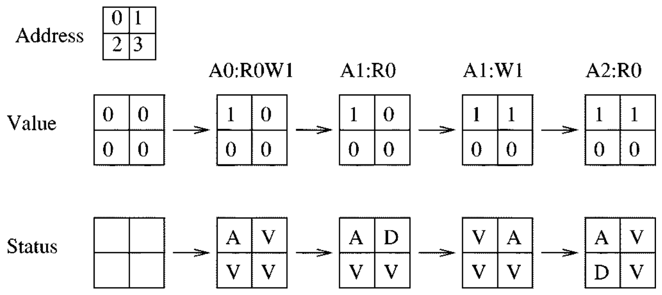


Fig. 7. Fault simulation example for CFin⟨$\uparrow;\updownarrow$⟩.

The simulation algorithm is simple and extensible. For each test operation, various operation flags are set. For example, the Write-1 (W1) flag is set for a Write-1 operation and the up transition (UTR) flag is set for a Write-1 operation only when it causes a 0 to 1 transition of a cell. There are more operation flags such as down transition (DTR), last-read-value (LRV), last-write-value (LWV), etc. These flags are updated for each test operation to record the current state of the cell. New flags can be easily added if it is necessary for a certain memory architecture.

After setting the operation flags, the attributes described in the fault descriptors are checked for the fault activation and fault detection conditions. If the AGR matches the operation flags, then the cell is in the aggressor mode and the victim flags should be set for all possible victims. The aggressor address is recorded by each victim. If the RCV matches in the operation flags, the victim flag and the aggressor entry are cleared for the memory cell. If the VTM matches in the operation flags, the fault effect is observable and it is marked as detected.

An example of the fault simulation algorithm in execution under a simple two-element test algorithm, for the fault CFin⟨$\uparrow;\updownarrow$⟩, is illustrated in Fig. 7. There are four memory cells under simulation, i.e., Cells 0, 1, 2, and 3. The initial background is all-0 after the first March element. The fault descriptor is shown in Table II. In the beginning of the second March element, a Read-0 is applied to Cell 0, followed immediately by a Write-1 to make an up transition in Cell 0. The cell is in the aggressor mode according to the fault descriptor, so RAMSES will set the victim flags of all other cells. The aggressor address is recorded by the victim cells. Next, a Read-0 is applied to Cell 1 and the Cell 0 to Cell 1 coupling fault is marked as detected by RAMSES. Note that the value of the cell is not changed to the faulty value and the fault detection condition is determined only by checking the flags. After that, a Write-1 is applied to Cell 1 and the up transition makes it an aggressor and its victim flag is cleared because of the Write-1. The victim flags are set for those that should and the victims record the aggressor address. In the final step shown in the figure, a Read-0 is applied to Cell 2 and two coupling faults are marked as detected, i.e., the Cell 0 to Cell 2 coupling and Cell 1 to Cell 2 coupling. The algorithm continues like this until all cells have been visited.

For word-oriented memories, single cell faults can be simulated in the same way, but the relative strength of the Write operation and the coupling effect should be defined first in order to simulate intraword coupling faults. If the Write operation is stronger than the coupling effect, then the coupling effect will be masked by the Write operation and the fault is a redundant fault. Therefore, intraword coupling faults can be detected only when the coupling effect is stronger than the Write operation. RAMSES deals with such coupling faults by disabling write recovery when the aggressor is in the same word as the victim.

For multiport memories, additional operation flags are required for interport faults. Port specification also is necessary for the operation flags. For example, a Read-0 operation must explicitly specify the port from which it reads—R0(A) specifies a Read-0 from port A. Other attributes follow the same rule.

IV. TAGS

The proposed test generator, TAGS [17], was developed based on the notion of a *March template*. A march template is defined as a sequence of Read–Write operations similar to a March test, but without the explicit specification of address sequences and data backgrounds. For example, $(w)(rw)(rwr)$ is a March template. The template consists of one or more template elements, such as (w) and (rw). From a March template, we can derive various March tests by applying different data and address sequence combinations. The FC of each test is calculated by RAMSES. Note that the generation of all possible March templates or March tests has an exponential complexity for both time and space. However, we have observed that most of the exhaustively generated March tests are inefficient and can be discarded. Test generation options and filtering conditions have been developed to greatly lower the complexity. We will introduce the TAGS algorithm for bit-oriented memories first, then the extended ones to handle word-oriented memories and multiport memories.

A. TAGS for Bit-Oriented Memories

The test generation procedure by TAGS is summarized as follows.

1) Initialize the test length as $1N$. Let the template set contain a single template $t = (w)$.
2) Increase the test length by $1N$: for each template t in the template set, add a Read/Write operation to t using one of the generation options (to be shown later). Repeat the step for all possible cases to form a new template set, except when any of the filtering conditions (to be shown later) is true.
3) A series of March tests is generated by assigning address orders in various combinations to each template in the set, together with consistent data backgrounds. When the address order for a stand-alone Read or Write can be either $\Uparrow$ or $\Downarrow$, we use $\Uparrow$ by default.
4) Simulate the resulting March tests using RAMSES.
5) Drop the tests which have no FC improvement over tests in the previous iteration or if the improvement is completely covered by another test in the current iteration.
6) Repeat Steps 2 to 5 using the new template set until the given fault set is 100% covered or the test length limit is reached.

The generation options are heuristics for generating effective March templates. Read/Write operations are inserted into an existing March template in many ways that can make it activate more memory faults and/or observe more errors. There are many possible combinations of March templates even with very simple generation options, therefore some filtering conditions are necessary for dropping the ineffective templates to reduce the time complexity. We now describe the generation options and filtering conditions that we apply for the fault models discussed in this paper. The generation options are as follows. Longer templates are derived from shorter ones by using only these options.

- Insert a stand-alone Read operation, i.e., (r), anywhere in t except in the beginning.
- Pick a template element and insert a Read operation in the beginning or append one at the end.
- Insert a stand-alone Write operation, i.e., (w), anywhere in t except in the beginning or at the end.
- Pick a template element and insert a Write operation in the beginning or append one at the end.

The filtering conditions are as follows. Ineffective templates are dropped if any of the conditions is true.

- There are three consecutive read operations (i.e., $(\cdots rrr \cdots)$) in a template element.
- There are three consecutive stand-alone read template elements (i.e., $\cdots (r)(r)(r) \cdots$) in the template.

We show a simple example to delineate the TAGS approach. Assume that before we get to Step 2), the template set contains only one template, i.e., $\{(w)(r)\}$. After we finish Step 2), the new template set is $\{(w)(wr), (w)(rr), (w)(rw), (w)(r)(r), (w)(w)(r)\}$. Then, after Step 3), the resulting March tests are as given in Table III. After the fault simulation by RAMSES, only three of these eight algorithms are selected (see Table IV) and all the others are dropped. The procedure is repeated for the new template set generated for each iteration.

The following example shows the complete result generated by TAGS. Given the target faults, SAF, TF, AF, SOF, CFin, CFst, CFid, and RDF and an unlimited test length, the March test algorithms generated by TAGS are shown in Table V, where M_i^j represents the ith test algorithm with complexity jN. As shown in the table, a series of tests are generated by TAGS, with increasing complexity (test length) and FC. The test generation

TABLE III
3N MARCH TESTS GENERATED BY TAGS AFTER STEP 3

No.	Test
1	$\Uparrow (w0) \Uparrow (w1, r1)$
2	$\Uparrow (w0) \Downarrow (w1, r1)$
3	$\Uparrow (w0) \Uparrow (r0, r0)$
4	$\Uparrow (w0) \Downarrow (r0, r0)$
5	$\Uparrow (w0) \Uparrow (r0, w1)$
6	$\Uparrow (w0) \Downarrow (r0, w1)$
7	$\Uparrow (w0) \Uparrow (r0) \Uparrow (r0)$
8	$\Uparrow (w0) \Uparrow (w1) \Uparrow (r1)$

TABLE IV
3N MARCH TESTS SELECTED BY RAMSES

No.	Test
3	$\Uparrow (w0) \Uparrow (r0, r0)$
5	$\Uparrow (w0) \Uparrow (r0, w1)$
8	$\Uparrow (w0) \Uparrow (w1) \Uparrow (r1)$

process stops at $12N$, when the FC reaches 100%, i.e., it returns a complete test M_1^{12}. The test is an irredundant March test for the above faults—dropping any operation or element causes an FC loss.

RAMSES simulation results for the first test algorithm in every pass j (i.e., M_1^j) are shown in Fig. 8. The tradeoff between test length and FC can be observed—in general, the overall FC increases as the test length increases, using the proposed approach. However, the FC for a particular fault may stay the same until a certain test element is added to the test, e.g., SOF can be detected by $\Updownarrow (r0, w1, r1)$ or $\Updownarrow (r1, w0, r0)$, so its FC is almost 0 until the inclusion of any of the above test elements, i.e., until the $12N$ algorithm is found. Of course a different strategy in TAGS can generate $\Updownarrow (r0, w1, r1)$ or $\Updownarrow (r1, w0, r0)$ in an earlier stage, but then the full coverage of some other faults will be delayed during the test generation process. Note that a Read from a cell with SOF will get the data from the previous Read [3], so it is clear that any of $\Updownarrow (r0, w1, r1)$ and $\Updownarrow (r1, w0, r0)$ can fully detect SOF. However, what was not clear but can be reported by TAGS is that some March elements other than $\Updownarrow (ra, wb, rb)$ will also detect a few SOFs. For example, the SOF of the first cell can be detected by the $r1$ operation of $\{\Updownarrow (r0, w1); \Uparrow (r1, w0)\}$. Therefore, the SOF coverage is very low but not zero before $\Updownarrow (r0, w1, r1)$ or $\Updownarrow (r1, w0, r0)$ is included in the test.

Fig. 8 shows only one test for each test length. In most cases, there is more than one test for a given test length. For example, in Fig. 9 we depict the FC numbers for all $8N$ tests. In this figure, e.g., M_2^8 detects 75% of CFid, because three out of the four March elements that are necessary for detecting CFid are part of the test. Only the March element $\Downarrow (r1, w0)$ is missing. As a second example, 87.5% of CFst is detected by M_2^8, with a 75% FC contributed by the same March elements that detect CFid and the other 12.5% contributed by the final $\Uparrow (r1)$ element. Again, the March element $\Downarrow (r1, w0)$ is missing in this test so far as complete detection is concerned. We can generate various tests of the same length with different fault detection capability. For example, if SAF, AF, SOF, and RDF are the most important fault models, then M_1^8 is the best $8N$ test. However, it is not good for

TABLE V
EXAMPLE TEST ALGORITHMS GENERATED BY TAGS

1N	M_1^1	$\Uparrow (w0)$
2N	M_1^2	$\Uparrow (w0) \Uparrow (r0)$
3N	M_1^3	$\Uparrow (w0) \Uparrow (w1) \Uparrow (r1)$
3N	M_2^3	$\Uparrow (w0) \Uparrow (r0, r0)$
3N	M_3^3	$\Uparrow (w0) \Uparrow (r0, w1)$
4N	M_1^4	$\Uparrow (w0) \Uparrow (r0) \Uparrow (r0, w1)$
4N	M_2^4	$\Uparrow (w0) \Uparrow (w1, r1) \Uparrow (r1)$
4N	M_3^4	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1)$
4N	M_4^4	$\Uparrow (w0) \Uparrow (r0, w1, r1)$
5N	M_1^5	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1)$
5N	M_2^5	$\Uparrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0)$
5N	M_3^5	$\Uparrow (w0) \Uparrow (w1) \Uparrow (r1, w0) \Uparrow (r0)$
5N	M_4^5	$\Uparrow (w0) \Uparrow (w1) \Uparrow (r1, w0, r0)$
6N	M_1^6	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0, r0)$
6N	M_2^6	$\Uparrow (w0) \Uparrow (r0) \Uparrow (r0, w1, r1) \Uparrow (r1)$
6N	M_3^6	$\Uparrow (w0) \Uparrow (r0) \Uparrow (r0, w1) \Downarrow (r1, w0)$
6N	M_4^6	$\Uparrow (w0) \Uparrow (w1) \Uparrow (r1, w0, r0) \Uparrow (r0)$
6N	M_5^6	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Downarrow (r1, w0)$
6N	M_6^6	$\Uparrow (w0) \Uparrow (w1, r1) \Uparrow (r1, w0, r0)$
6N	M_7^6	$\Uparrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0) \Uparrow (r0)$
6N	M_8^6	$\Uparrow (w0) \Uparrow (r0, w1, r1, w0) \Uparrow (r0)$
6N	M_9^6	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0)$
6N	M_{10}^6	$\Uparrow (w0) \Uparrow (w1, r1) \Uparrow (r1, w0) \Uparrow (r0)$
6N	M_{11}^6	$\Uparrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0, r0)$
7N	M_1^7	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Downarrow (r0, w1)$
7N	M_2^7	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0) \Uparrow (r0)$
7N	M_3^7	$\Uparrow (w0) \Uparrow (w1, r1) \Uparrow (r1, w0, r0) \Uparrow (r0)$
7N	M_4^7	$\Uparrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0, r0) \Uparrow (r0)$
7N	M_5^7	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0, r0)$
7N	M_6^7	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1) \Downarrow (r1, w0) \Uparrow (r0)$
7N	M_7^7	$\Uparrow (w0) \Uparrow (r0) \Uparrow (r0, w1, r1) \Downarrow (r1, w0)$
7N	M_8^7	$\Uparrow (w0) \Uparrow (r0) \Uparrow (r0, w1) \Uparrow (r1) \Downarrow (r1, w0)$
7N	M_9^7	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (w0) \Downarrow (r0, w1, r1)$
8N	M_1^8	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1) \Downarrow (r1, w0, r0) \Uparrow (r0)$
8N	M_2^8	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Downarrow (r0, w1) \Uparrow (r1)$
8N	M_3^8	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0, r0) \Uparrow (r0)$
8N	M_4^8	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0) \Downarrow (r0, w1)$
8N	M_5^8	$\Uparrow (w0) \Uparrow (w1, r1) \Uparrow (r1, w0) \Uparrow (r0) \Downarrow (r0, w1)$
9N	M_1^9	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Downarrow (r0, w1) \Downarrow (r1, w0)$
9N	M_2^9	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0) \Downarrow (r0, w1) \Uparrow (r1)$
9N	M_3^9	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0) \Uparrow (r0) \Downarrow (r0, w1)$
10N	M_1^{10}	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Downarrow (r0, w1, r1) \Downarrow (r1, w0)$
10N	M_2^{10}	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Downarrow (r0, w1) \Downarrow (r1, w0) \Uparrow (r0)$
10N	M_3^{10}	$\Uparrow (w0) \Uparrow (r0, r0, w1, r1) \Uparrow (r1, w0) \Downarrow (r0, w1) \Uparrow (r1)$
11N	M_1^{11}	$\Uparrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0) \Downarrow (r0, w1) \Downarrow (r1, w0) \Uparrow (r0)$
11N	M_2^{11}	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, r1, w0, r0) \Downarrow (r0, w1) \Downarrow (r1, w0)$
12N	M_1^{12}	$\Uparrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0, r0) \Downarrow (r0, w1, r1) \Downarrow (r1, w0)$ $\Uparrow (r0)$

CFst or CFin as compared with other $8N$ tests. This information is especially valuable when a 100% test of all faults is not affordable or necessary. Selection of a test algorithm from the set can be based on the priority of faults that are considered, which is process and product dependent. Note also that the real test time can vary even for algorithms with the same complexity, since different memories have different Read/Write-mode implementations.

B. TAGS for Word-Oriented Memories

A word-oriented memory has Read/Write operations that access the memory array by a word, instead of by a bit. Word-oriented memories can be tested by applying a bit-oriented test algorithm repeatedly with a set of different data backgrounds [3], [5], [13]. The repeating procedure multiplies the testing time. For example, three different data backgrounds, i.e., 0000, 0101, and 0011, are required to test a 4-bit word-oriented memory. When using a $10N$ test algorithm, the testing time will be $30N$. We have shown in [9] that testing word-oriented memories by

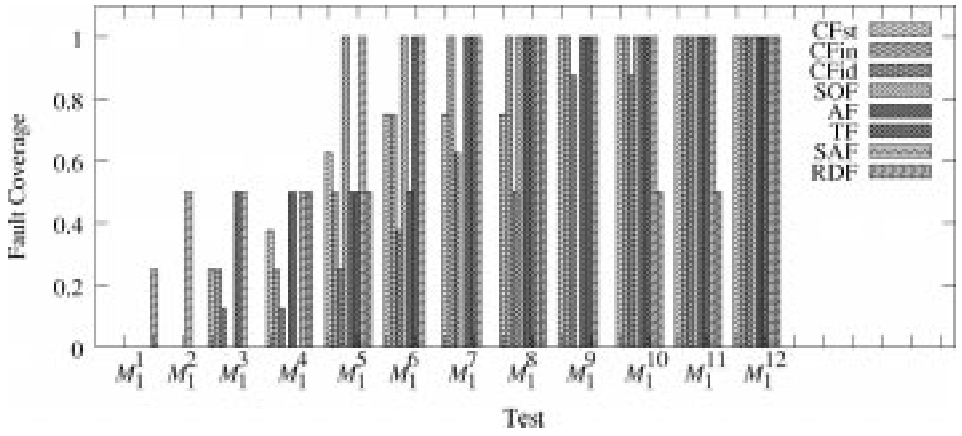


Fig. 8. RAMSES simulation results for M_1^j.

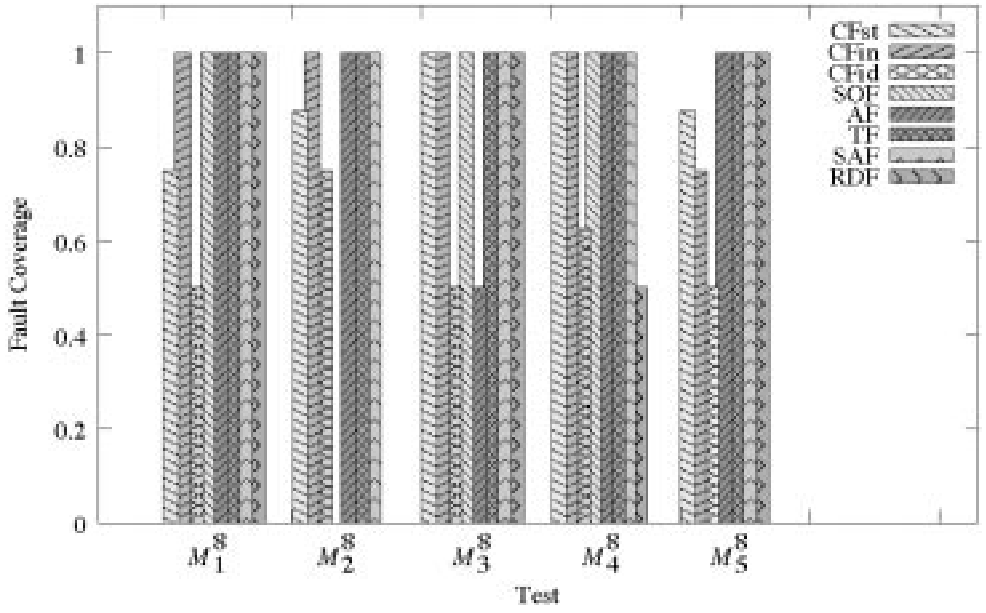


Fig. 9. Fault coverage simulation results for the $8N$ tests.

repeatedly applying a single March test with different data backgrounds is not cost-effective. Most faults are covered by the test even with only a single data background. Additional test runs with other data backgrounds only covers a small number of additional faults, e.g., intraword coupling faults.

The *Cocktail–March* algorithms [10] are a class of more efficient March tests for word-oriented memories. We have extended TAGS for word-oriented memories based on the Cocktail–March algorithms; by mixing different test algorithms and different data backgrounds, the overall test length can be significantly reduced. For a target fault set, the steps for TAGS to generate the test algorithms are as follows.

1) Construct the bit-oriented memory test algorithms using the procedure presented in Section IV-A
2) Generate the initial Cocktail–March test (assuming the word length is m).
   a) Generate a set of data backgrounds $P = \{p_0, p_1, \ldots, p_K\}$, where $K = \lceil \log_2 w \rceil$. For $1 \leq j \leq K$, the background word p_j is represented as $p_j = b_{m-1} \cdots b_1 b_0$, where $b_i = 1$ if $i \bmod 2^j \geq 2^{j-1}$ and $b_i = 0$ otherwise. Table VI shows an example for 8-bit backgrounds, where the all-zero background is also called the *solid background*.
   b) Assign each and every data background, one by one, to the complete test algorithm generated in Step 1) (which is the initial *candidate* test algorithm), as in the traditional method, resulting in a cascade of multiple March algorithms.
3) Optimize the Cocktail–March test (which is now a cascade of multiple March algorithms) for each p_j, except p_0.
   a) Generate a new Cocktail–March test by replacing the March algorithm having p_j as its background with a shorter one from the set of algorithms generated in Step 1).
   b) Run RAMSES for the new Cocktail–March.
   c) Repeat Steps 3a) and 3b) until the FC drops and cannot be recovered by any other test algorithm of the same length.
   d) Store the candidate test algorithms used in the previous step.

TABLE VI
8-BIT DATA BACKGROUNDS

p_j	$b_7b_6b_5b_4b_3b_2b_1b_0$
p_0	00000000
p_1	01010101
p_2	00110011
p_3	00001111

TABLE VII
INITIAL COCKTAIL–MARCH TEST

Background	p_0	p_1	p_2	p_3
Candidates	M_1^{12}	M_1^{12}	M_1^{12}	M_1^{12}

4) Optimize the Cocktail–March test from the previous step for p_0 (the solid background).
   a) Generate a new Cocktail–March test by replacing the March algorithm having p_0 as its background with a shorter one from the set of test algorithms generated in Step 1). Repeat with every test candidate for other backgrounds.
   b) Run RAMSES for the new Cocktail–March.
   c) Repeat Steps 4a) and 4b) for all candidate test algorithms from Step 3d) until the FC drops and cannot be recovered by any other test algorithm of the same length or by selecting other candidates.

We now give an example to illustrate the test generation procedure. Again, the target fault models are SAF, TF, AF, SOF, CFin, CFst, CFid, and RDF. The memory under test is an 8-bit word-oriented memory. After Step 1), a set of test algorithms is generated as in Table V. Step 2) generates a set of data backgrounds as shown in Table VI then initializes the Cocktail–March by assigning the bit-oriented complete test, i.e., M_1^{12}, to all data backgrounds. The initial $48N$ test algorithm is shown in Table VII. We optimize it for p_1 by replacing M_1^{12} with M_1^{11}, M_2^{11}, M_1^{10}, and M_2^{10}, one by one and calculate the FC values by RAMSES. This optimization procedure stops when the test length drops to $5N$, when we observe the occurrence of an FC drop. We repeat the optimization step for p_2 and p_3, respectively. For each of p_1, p_2, and p_3, usable $5N$ candidate tests are the same, i.e., M_3^5 and M_4^5, as shown in Table VIII. In Step 4), the test algorithm for the solid background is optimized by the selection of candidate tests for p_1, p_2, and p_3. Finally, the optimized Cocktail–March test is generated, as shown in Table IX. Our Cocktail–March test reduces the test length from $48N$ to $27N$ in this case, i.e., a 43.7% test time reduction.

In this case, single cell faults and interword related faults are already covered by the bit-oriented test using just the solid background. Additional backgrounds only cover a small number of extra intraword faults. Fig. 10 shows the FC improvement of intraword coupling faults with respect to the test length, while the FC for other faults is already 100%. Note that a similar observation can be derived from [13]. However, the test of intraword related faults other than the conventional faults is difficult to find manually. The proposed fault simulator makes automatic test generation easy. With RAMSES, the coverage of even untargeted faults also can evaluated and instead of "detected or not," the FC figures are accurately reported.

TABLE VIII
COCKTAIL–MARCH ALGORITHM DURING OPTIMIZATION

Background	p_0	p_1	p_2	p_3
Candidates	M_1^{12}	$M_3^5\ M_4^5$	$M_3^5\ M_4^5$	$M_3^5\ M_4^5$

TABLE IX
FINAL COCKTAIL–MARCH ALGORITHM

Background	Test
$p_0(00000000)$	$\Uparrow(wa)\Uparrow(ra,w\bar{a},r\bar{a})\Uparrow(r\bar{a},wa,ra)$ $\Downarrow(ra,w\bar{a})\Downarrow(r\bar{a},wa)\Uparrow(ra)$
$p_1(01010101)$	$\Uparrow(wa)\Uparrow(w\bar{a})\Uparrow(r\bar{a},wa,ra)$
$p_2(00110011)$	$\Uparrow(wa)\Uparrow(w\bar{a})\Uparrow(r\bar{a},wa,ra)$
$p_3(00001111)$	$\Uparrow(wa)\Uparrow(w\bar{a})\Uparrow(r\bar{a},wa,ra)$

For comparison, we now present two more cases for 8-bit memories. In Case 1, the target faults are SAF, AF, SOF, RDF, and CFst and the TAGS results are listed in Table X. Note that with TAGS, test generation for different target faults is fast. For example, in Case 2 the target faults are SAF, TF, AF, SOF, and CFid and the generated test algorithm is shown in Table XI. Moreover, the test coverage can easily be evaluated by RAMSES (see Table XII). When there are several test algorithms generated by TAGS, the coverage of untargeted faults can be considered for algorithm selection.

C. *TAGS for Multiport Memories*

For multiport memories, the complexity of interport short fault increases with the number of ports. Several March-like test algorithms for multiport memories have been proposed in [18], [19], [21], and [22]. The March 2PF algorithm proposed in [15] detects some complex multiport faults. The test complexity, however, is high (e.g., $269N$) even when the scrambling information is available. On the other hand, it has been shown feasible to modify existing March algorithms to generate simple and efficient tests with much lower complexity [19], provided that the layout topology is available. We will show that with TAGS it is possible to extend March algorithms to cover multiport faults in a systematic way and test generation, test application, and diagnosis can be made easy. We will present the test generation approach for a two-port RAM, which can be extended to more-port ones easily.

As we have discussed in Section II, for a two-port RAM, the number of all possible interport shorts is $O(N^3)$. When the layout topology is not available, though we need to consider the entire fault set, the test time complexity is $O(N^2)$. As illustrated in Fig. 2, for each and every address accessed through Port A, a test that can detect all address decoder faults needs to be applied to Port B. Since the test for single-port address decoder faults is $O(N)$, the time complexity for testing all interport shorts is $O(N^2)$, which is still high and may not be affordable in practice. However, when the layout topology is known, then possible shorts can be identified and used to derive a test with lower time complexity. Assume the interport shorts can only exist in between two adjacent rows (word-lines) or two adjacent columns (bit-lines), then a linear-time test algorithm can be found. We begin by introducing the March guards.

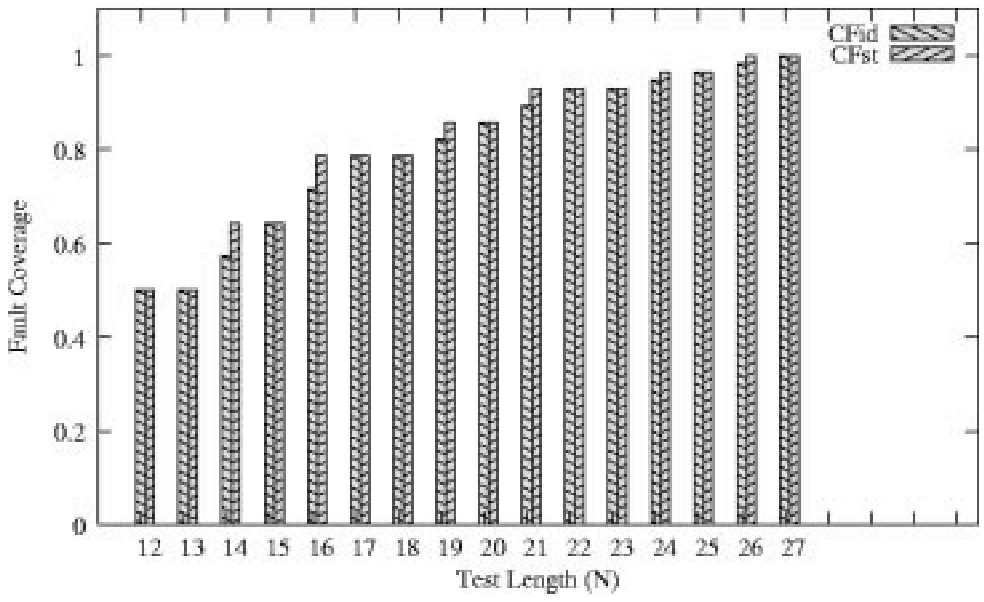


Fig. 10. Intraword fault coverage improvement.

TABLE X
CASE 1 TEST—FOR SAF, AF, SOF, RDF, AND CFst

Background	Test
$p_0(00000000)$	$\Uparrow(wa)\ \Uparrow(ra)\ \Downarrow(ra,w\bar{a})\ \Uparrow(wa)$ $\Uparrow(ra,w\bar{a},r\bar{a})\ \Uparrow(r\bar{a},wa)$
$p_1(01010101)$	$\Uparrow(wa)\ \Uparrow(ra,w\bar{a},r\bar{a})$
$p_2(00110011)$	$\Uparrow(wa)\ \Uparrow(ra,w\bar{a},r\bar{a})$
$p_3(00001111)$	$\Uparrow(wa)\ \Uparrow(ra,w\bar{a},r\bar{a})$

TABLE XI
CASE 2 TEST—FOR SAF, TF, AF, SOF, AND CFid

Background	Test
$p_0(00000000)$	$\Uparrow(wa)\ \Uparrow(ra,w\bar{a})\ \Uparrow(r\bar{a},wa)\ \Downarrow(ra,w\bar{a})$ $\Downarrow(r\bar{a},wa)\ \Uparrow(ra)$
$p_1(01010101)$	$\Uparrow(wa)\ \Uparrow(w\bar{a})\ \Uparrow(r\bar{a},wa,ra)$
$p_2(00110011)$	$\Uparrow(wa)\ \Uparrow(w\bar{a})\ \Uparrow(r\bar{a},wa,ra)$
$p_3(00001111)$	$\Uparrow(wa)\ \Uparrow(w\bar{a})\ \Uparrow(r\bar{a},wa,ra)$

TABLE XII
FC COMPARISON BETWEEN CASE 1 AND CASE 2 TESTS, FOR A $1K$ RAM

	SAF	TF	SOF	RDF	CFin	CFst	CFid	AF
Case 1	1	1	1	1	1	1	0.7970	1
Case 2	1	1	1	0.0	1	1	1	1

For each cell (called the base cell), four Read operations are defined on its north, south, east, and west neighboring cells, denoted as r^N, r^S, r^E, and r^W, respectively. These four Read operations are called the *March guards* of the base cell. When the base cell is a boundary cell, a March-guard Read can be either degenerated into a No-Operation (NOP), or reading the base cell itself, depending on the implementation preference. A $7N$ March test extended with the March guards for a two-port memory is shown in Fig. 11. The example March algorithm is in fact the MATS++ algorithm with an additional $r0$ operation in the last element (i.e., M_2), which is extended to test two-port memories by adding the March guards. The test algorithm is executed as a normal March test during the first two elements,

	M_0	M_1	M_2
Port A	$\Updownarrow(w0)$	$\Uparrow(r0,w1)$	$\Downarrow(r1,w0,r0,r0)$
Port B	$(-)$	$(-,-)$	$(r^{\mathrm{N}}1,r^{\mathrm{S}}0,r^{\mathrm{W}}1,r^{\mathrm{E}}0)$

Fault Coverage		
Inter-port WSF	OR-type	63.7%
	AND-type	31.8%
Inter-port BSF	OR-type	63.7%
	AND-type	31.8%

Fig. 11. An extended March algorithm with March guards for interport short faults.

M_0 and M_1, through Port A. Meanwhile, Port B is kept quiet (denoted by "—"). During element M_2, both Ports A and B are tested concurrently. A $4N$ test element is applied to Port A and the March guards are applied to Port B. Note that the March element for Port B follows the same address order as the corresponding element for Port A.

With March guards, TAGS can be extended to generate test algorithms for multiport memories (not just two-port ones). The following procedure shows the steps of TAGS to generate tests for a two-port memory, with Ports A and B. Assume that when accessing the same address, Port A is given precedence over Port B and all cell faults are detectable by either Port A or B. The target single-port fault models are SAF, TF, AF, SOF, CFin, CFst, CFid, and RDF and the target interport fault models are WSF and BSF. Fig. 12 shows the resulting $14N$ test algorithm after TAGS finished all the steps. The algorithm detects 100% of the target faults.

1) Generate a complete test for the target single-port faults, resulting in a $12N$ test, denoted as M_1^{12}. The test algorithm is assigned to Port A.
2) Select two elements from M_1^{12} such that one of them has an ascending address order and the other has a descending address order (longer test elements have higher priority) and add March guards to Port B in the corresponding elements.

	M_0	M_1	M_2
Port A	$\Updownarrow(w0)$	$\Uparrow(r0, w1)$	$\Uparrow(r1, w0, r0, r0)$
Port B	$(-)$	$(r0, r1)$	$(r^{\mathrm{N}}1, r^{\mathrm{S}}0, r^{\mathrm{W}}1, r^{\mathrm{E}}0)$

	M_3	M_4	M_5
Port A	$\Downarrow(r0, w1, r1, r1)$	$\Downarrow(r1, w0)$	$\Updownarrow(r0)$
Port B	$(r^{\mathrm{S}}0, r^{\mathrm{N}}1, r^{\mathrm{E}}0, r^{\mathrm{W}}1)$	$(r1, r0)$	$(r0)$

Fault Coverage		
SAF, TF, AF, SOF, CFin, CFst, CFid, RDF		100%
Inter-port WL short	Or-type	100%
	And-type	100%
Inter-port BL short	Or-type	100%
	And-type	100%

Fig. 12. March guard extension for M_1^{12}.

If the selected element has less than four operations (i.e., it is shorter than $4N$), then extra Read operations are added to make it a $4N$ element.

Record the FC for WSF and BSF.

3) Repeat the last step for all possible element pairs and March guard orders until the FC of interport shorts reaches 100%.
4) Change the NOPs in Port B to Read operations (the same with the corresponding Port A operations), except for the initial Write operation.

In summary, TAGS first generates the *base* algorithm, e.g., M_1^{12}, that covers most faults in the memory. The base algorithm then can be easily extended to test specific memory architectures and cover additional fault models. The test generation procedure for word-oriented memories and multiport memories were presented to illustrate the basic idea of how extensions can be done to TAGS. There are other important issues that are not included in our discussion here, e.g., complex fault models such as shorts between word lines and bit lines [22], test scheduling for multiport memories to further reduce the test time [20], etc. However, following our simulation-based technique, generalization of the approach to cover other memory types and fault models can be done without much effort.

V. Discussion and Conclusion

The test length of word-oriented Cocktail–March in Section IV is $(12 + 5\log_2 B)N$ when the target faults include SAF, TF, AF, SOC, CFin, CFst, CFid, and RDF, where B is the word length and N is the address count (address space) for word-oriented memories. As a comparison, in [13] the length of the test algorithm based on March C- for intraword CFid, CFst, and CFdst is $(10 + 6\log_2 B)N$. The $10N$ base algorithm they use is different simply because of the different set of target faults. Note that by TAGS, the length of the additional test for covering the intraword faults is $(5\log_2 B)N$, while the length of the extra test for covering the intraword faults is $(6\log_2 B)N$ by that reported in [13]. Note also that in our case, though the specified intraword faults does not include CFdst, the generated test covers CFdst as well.

In conclusion, novel and efficient fault simulation and test algorithm generation algorithms for random access memories have been presented. Our simulation-based test generation approach reduces the memory test development and evaluation effort significantly. It is flexible and can be generalized to cover other memory types and fault models easily. We have also implemented the software tools, RAMSES and TAGS and given experimental results for bit-oriented, word-oriented, and multiport memories. Efficient linear-time test algorithms have been obtained by our tools that cover all the target faults. By RAMSES and TAGS we have developed the Cocktail–March algorithms, a class of March tests for word-oriented memories which are the most efficient so far.

References

[1] A. K. Sharma, *Semiconductor Memories: Technology, Testing and Reliability*. New York: IEEE, 1997.

[2] B. Prince, *Semiconductor Memories: A Handbook of Design, Manufacture and Application,* 2nd ed. New York: Wiley, 1991.

[3] R. Dekker, F. Beenker, and L. Thijssen, "Fault modeling and test algorithm development for static random access memories," in *Proc. Int. Test Conf. (ITC)*, 1988, pp. 343–352.

[4] B. Nadeau-Dostie, A. Silburt, and V. K. Agarwal, "Serial interface for embedded-memory testing," *IEEE Design Test Computers*, vol. 7, pp. 52–63, Apr. 1990.

[5] A. J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*. Chichester, U.K.: Wiley, 1991.

[6] B. F. Cockburn, "Tutorial on semiconductor memory testing," *J. Electron. Testing: Theory Applicat.*, vol. 5, pp. 321–336, 1994.

[7] M. Riedel and J. Rajski, "Fault coverage analysis of RAM test algorithms," in *Proc. IEEE VLSI Test Symp. (VTS)*, 1995, pp. 227–234.

[8] E. Simonse, "Circuit structures, design requirements and fault simulations for CMOS SRAM's," Master's dissertation, Faculty of Information Technology and Systems, Delft Univ. Technol., 1998.

[9] C.-T. Huang, J.-R. Huang, C.-F. Wu, C.-W. Wu, and T.-Y. Chang, "A programmable BIST core for embedded DRAM," *IEEE Design Test Computers*, vol. 16, pp. 59–70, Jan.–Mar. 1999.

[10] C.-F. Wu, C.-T. Huang, and C.-W. Wu, "RAMSES: A fast memory fault simulator," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT)*, Albuquerque, NM, Nov. 1999, pp. 165–173.

[11] A. J. van de Goor and B. Smit, "Generating march tests automatically," in *Proc. Int. Test Conf. (ITC)*, 1994, pp. 870–878.

[12] ——, "Automating the verification of memory tests," in *Proc. IEEE VLSI Test Symp. (VTS)*, 1994, pp. 312–318.

[13] A. J. van de Goor and I. B. S. Tlili, "March tests for word-oriented memories," in *Proc. Design, Automation Test in Europe (DATE)*, 1998, pp. 501–508.

[14] K. Zarrineh, S. J. Upadhyaya, and S. Chakravarty, "A new framework for generating optimal march tests for memory arrays," in *Proc. Int. Test Conf. (ITC)*, 1998, pp. 73–82.

[15] A. J. van de Goor and S. Hamdioui, "Fault models and tests for two-port memories," in *Proc. IEEE VLSI Test Symp. (VTS)*, 1998, pp. 401–410.

[16] A. J. van de Goor, "Using march tests to test SRAM's," *IEEE Design Test Computers*, vol. 10, pp. 8–14, Mar. 1993.

[17] C.-F. Wu, C.-T. Huang, K.-L. Cheng, and C.-W. Wu, "Simulation-based test algorithm generation for random access memories," in *Proc. IEEE VLSI Test Symp. (VTS)*, Montreal, Canada, Apr. 2000, pp. 291–296.

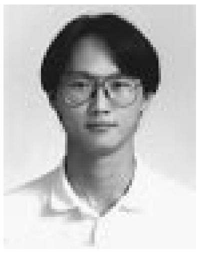
[18] Y. Wu and S. Gupta, "Built-in self-test for multi-port RAM's," in *Proc. Sixth IEEE Asian Test Symp. (ATS)*, 1997, pp. 398–403.

[19] J. Zhao, S. Irrinki, M. Puri, and F. Lombardi, "Detection of inter-port faults in multi-port static RAM's," in *Proc. IEEE VLSI Test Symp. (VTS)*, 2000, pp. 297–302.

[20] C.-F. Wu, C.-T. Huang, K.-L. Cheng, C.-W. Wang, and C.-W. Wu, "Simulation-based test algorithm generation and port scheduling for multi-port memories," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, Las Vegas, NV, June 2001, pp. 301–306.

[21] T. Matsumura, "An efficient test method for embedded multi-port RAM with BIST circuitry," in *Proc. IEEE Int. Workshop Memory Technology, Design and Testing (MTDT)*, San Jose, CA, 1995, pp. 62–67.

[22] F. Karimi, S. Irrinki, T. Crosby, and F. Lombardi, "A parallel approach for testing multi-port static random access memories," in *Proc. IEEE Int. Workshop on Memory Technology, Design and Testing (MTDT)*, San Jose, CA, Aug. 2001, pp. 73–81.



Chi-Feng Wu (S'98) received the B.S. degree in 1996, the M.S. degree in 1998, and the Ph.D. degree in 2001, all in electrical engineering, from National Tsing Hua University (NTHU), Hsinchu, Taiwan, R.O.C.

His research interests include the design and test of VLSI cores and systems. He is currently an Engineer at Realtek Semiconductor Corp., Hsinchu, Taiwan, R.O.C., investigating reusable IP design, system-on-chip integration, and testing.

Chih-Tsun Huang (S'98–M'01) received the B.S. degree in 1994, the M.S. degree in 1996, and the Ph.D. degree in 2000, all in electrical engineering, from National Tsing Hua University (NTHU), Hsinchu, Taiwan, R.O.C.

His research areas include VLSI design and test, design for testability, SOC testing, as well as embedded memory testing. He is also interested in reliable design of VLSI circuits, architectures, and core-based system chips. He is currently a postdoc in the Design Technology Center of NTHU and an Adjunct Assistant Professor of the Department of Electrical Engineering, NTHU.

Kuo-Liang Cheng (S'01) received the B.S. degree, in 1998, and the M.S. degree, in 1999, from the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, R.O.C., where he is currently pursuing the Ph.D. degree.

His research interests include VLSI design and testing, as well as the testing and diagnosis of semiconductor memory.

Cheng-Wen Wu (S'86–M'87–SM'95) received the BSEE degree, in 1981, from National Taiwan University, Taipei, Taiwan, R.O.C., and the M.S. and Ph.D. degrees, both in electrical and computer engineering, in 1985 and 1987, respectively, from the University of California, Santa Barbara.

From 1981 to 1983, he was an Ensign Instructor at the Chinese Naval Petty Officers' School of Communications and Electronics, Tsoying, Taiwan, R.O.C. From 1983 to 1984, he was with the Information Processing Center of the Bureau of Environmental Protection, Executive Yuan, Taipei, Taiwan, R.O.C. From 1985 to 1987, he was a Post Graduate Researcher at the Center for Computational Sciences and Engineering at UCSB. Since 1988, he has been with the Department of Electrical Engineering, National Tsing Hua University (NTHU), Hsinchu, Taiwan, R.O.C., where he is currently a Professor. He also has served as the Director of the university's Computer and Communications Center, from 1996 to 1998, and the Director of the university's Technology Service Center, from 1998 to 1999. From August 1999 to February 2000, he was a Visiting Faculty Member of the ECE Department, UCSB. Since August 2000, he has been the Chair of the Electrical Engineering Department of NTHU. He also is the Director of the IC Design Technology Center of the university. His research interests include design and testing of high performance VLSI circuits and systems.

Dr. Wu was the Technical Program Chair of the IEEE Fifth Asian Test Symposium (ATS'96) and the General Chair of ATS'00. He is an Associate Editor for the *Journal of the Chinese Institute of Electrical Engineers* (JCIEE) and a Guest Editor of the *JCIEE Special Issue on Design and Test of System-on-Chip*, and was a Guest Editor of the *Journal of Information Science and Engineering* (JISE), Special Issue on VLSI Testing. He received the Distinguished Teaching Award from NTHU in 1996, the Outstanding Electrical Engineering Professor Award from the Chinese Institute of Electrical Engineers (CIEE) in 1997, and the Distinguished Research Award from National Science Council in 2001. Dr. Wu is a life member of CIEE.