POLITECNICO DI TORINO
Repository ISTITUZIONALE

Automatic March Tests Generations for Static Linked Faults in SRAMs

Original
Automatic March Tests Generations for Static Linked Faults in SRAMs / Benso, Alfredo; Bosio, Alberto; DI CARLO, Stefano; DI NATALE, Giorgio; Prinetto, Paolo Ernesto. - STAMPA. - 1:(2006), pp. 1-6. (Intervento presentato al convegno Design, Automation and Test in Europe, Conference and Exhibition (DATE) tenutosi a Munich, DE nel 6-10 Mar. 2006) [10.1109/DATE.2006.244097].

Availability:
This version is available at: 11583/1499961 since:

Publisher:
IEEE Computer Society

Published
DOI:10.1109/DATE.2006.244097

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

12 October 2024

Automatic March Tests Generations for Static Linked Faults in SRAMs

A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
E-mail {benso, bosio, dicarlo, dinatale, prinetto}@polito.it
http://www.testgroup.polito.it

Abstract

Static Linked Faults are considered an interesting class of memory faults. Their capability of influencing the behavior of other faults causes the hiding of the fault effect and makes test algorithm design a very complex task. A large number of March Tests with different fault coverage have been published and some methodologies have been presented to automatically generate March Tests. In this paper we present an approach to automatically generate March Tests for Static Linked Faults. The proposed approach generates better test algorithms then previous, by reducing the test length

1. Introduction

Memories are one of the most important components in digital systems, and semiconductor memories are nowadays one of the fastest growing technologies. Actually the major trend of *System-On-a-Chip* (SOC) allows to embed in a single chip all the components and functions that historically were placed on a hardware board. Within SOCs, [1] forecasts that embedded memories will be the densest components, reaching the 90% of chips area surface in ten years. It is thus common finding, on a single chip, tens of memories of different types, sizes, access protocols and timing. Moreover they can recursively be embedded in embedded cores.

The high density of their cells array makes memories extremely vulnerable to physical defects. Due to the complex nature of the internal behaviour of memory chips, the design of fault models and tests is non-trivial.

An important class of memory faults is the class of *linked faults* [10]. A linked fault is a memory fault composed of two or more simple faults. The behaviour of each simple fault can be influenced by the remaining ones and in some cases the fault can be masked. Classic march tests cannot detect linked faults due to the masking. In the latest decade published researches mainly focused on the definition of new realistic fault models [2] [3] [4] [5] showing the importance of developing new memory test algorithms. Nevertheless few publications targeted the problem of linked faults and their *test generation*. In [6], [7], and [8] the authors present march test solutions having high fault coverage on a restricted set of linked memory faults. In [9] and [10] the authors present an accurate analysis of the linked fault concept. Moreover, they present a march test facing new fault models.

In [11], the authors present an automatic generation methodology, but it is still affected by the problem of considering a limited number of memory faults, the same of [6], [7], and [8]. So far, no other Automatic Test Generations for Static Linked Faults have been proposed.

In this paper we propose a new approach to automatically generate *March Tests* targeting static linked memory faults. The generation process resorts to a formal notation that extends the Fault Primitive notation [12] to describe the list of faulty behaviors to be detected. We successfully applied the proposed algorithm to an extensive set of realistic static linked faults, obtaining new March Tests, with a computation time in order of few seconds. To prove their correctness, all generated Tests have been fault simulated by an in-house developed memory fault simulator [13].

The paper is structured as follows: Section 2 presents an overview about memory fault modeling and notation. Section 3 introduces the concept of Linked Fault (LF) and extends the adopted formalism in order to model LFs. Section 4 presents the memory model. Section 5 details the steps of the automatic March Test generation process, whereas Section 6 presents experimental results. Section 7 finally

summarizes the main contributions and outlines future research activities.

2. Fault modeling

A *Functional Fault Model* (FFM) is a deviation of the memory behavior from the expected one under a set of performed operations. A FFM involves one or more *Faulty Memory Cells* (*f*-cells) classified in two categories: *Aggressor cells* (*a*-cells), i.e., the memory cells that sensitize a given FFM and *Victim cells* (*v-cells*), i.e., the memory cells that show the effect of a FFM. Each faulty behavior is sensitized by a sequence of *stimuli* applied on the *f*-cells. When dealing with SRAMs, the applied stimuli are the memory operations. First of all we have to specify the *initial conditions* of the cell, i.e. the value (state) of the memory cell, where we are going to apply the operations. Hereinafter we resort to *n* as the size of the memory (i.e., the number of memory cells)

Definition 1: *C* is the set of the memory states (values), formalized as

$$C = \{0^{[i]}, 1^{[i]}, -^{[i]} \mid 0 \le i \le n\text{-}1\} \qquad (1)$$

where the apex identifies the address of the cell. If the address is omitted, it means that the state can be applied on every memory cell indifferently. The '-' denotes a *don't care* condition.

Definition 2: *X* is the set of the memory operations, formalized as

$$X = \{r^{[i]}_{[d]}, w^{[i]}_{d} \mid 0 \le i \le n\text{-}1;\ d \in (0,1)\} \cup \{t\} \qquad (2)$$

where:

- w^i_d is a *write* operation of the value *d* performed in the cell *i;*
- r^i_d is a *read* operation performed in the cell *i*. The value *d* is not strictly needed in case of a read operation. If used, it means the expected value that should be red from the *i*-th memory cell;
- *t* is a *wait* operation for a defined period of time. This additional element is needed to deal with *Data Retention Faults* [14].

If the address is omitted, it means that the operation can be applied on every memory cell indifferently.

Each FFM can be described by a set of Fault Primitives (FPs) [12].

Definition 3: A *Fault Primitive* FP represents the difference between an expected (fault-free) and the observed (faulty) memory behavior denoted by:

$$< S_a\ ;\ S_v\ /\ F\ /\ R > \qquad (3)$$

Where S_a and S_v are the *Sequence of Sensitizing Operations and/or Conditions* respectively applied to *a*-cell and *v*-cell, needed to sensitize the given fault. The *j*-th condition/operation is represented as $c[x]$, where $c \in C$ (1), and $x \in X$ (2). $R = \{ (r)^n \mid r \in C \}$ is the sequence of values read on the aggressor cell when applying *S*.

As an example FP = $< 0w_1\ ;\ 0\ /\ 1\ /\ - >$ means that the operation 'w_1' performed on the *a*-cell, when the initial state is 0 for both a and v cells, causes the *v*-cell to flip. No addresses are specified; therefore this fault can affect each couple of memory cell. Several FPs classification rules can be adopted, based on the number of memory operations (*m*) needed to sensitize the FP (static when *m* = 1 or dynamic fault elsewhere); and based on the number of memory cells (#FC) involved by the FP (single-cell where #FC = 1 or n-cells elsewhere fault) [14]. Since the FP notation not necessarily explicates the address of both aggressor and victim memory cells, we extend the FP model by introducing the *Addressed Fault Primitive* concept.

Definition 4: An *Addressed Fault Primitive* (AFP) is an instantiation of a FP which explicit the involved addresses, and both the faulty and fault-free final memory state, reached by the memory, after applying the AFP. It can be formalized as:

$$AFP = (I\ ,\ E_s\ ,F_v\ ,\ G_v) \qquad (4)$$

where:

- $I = \{ (s)^{\#IC} \mid s \in C \}$ is the initial state, i.e., the value stored in the #IC involved cells, before applying the AFP. The first value correspond to the less significative bit (i.e. the memory cell with the lowest address);
- $E_s = \{ (op)^m \mid op \in X \}$ is the sequence of operations, performed on the aggressor cells, needed to sensitize the fault; each operation belong to the alphabet X, the set of all the possible memory operations;
- $F_v = \{ (f)^{\#IC} \mid f \in C \}$ is the logical value stored in the memory cells after applying Es (faulty state)
- $G_v = \{ (g)^{\#IC} \mid g \in C \}$ is the logical value stored in the memory cells after applying E_s on the fault-free memory (expected state).

The FP of the above example $< 0w_1\ ;\ 0\ /\ 1\ /\ - >$ can be translated into AFP1 = (00, w^0_1, 11, 10) and AFP2 =

(00, w^1_1, 11, 01), with a memory having $n = 2$ (i.e., two cells). Each AFP can be covered by a Test Pattern.

Definition 5: A *Test Pattern* for a given Addressed Fault Primitive is defined as:

$$TP = (I, E, O) \quad (5)$$

where:

- *I:* is the initial state (see 4);
- *E*: is equal to E_s (see 4);
- $O = \{r^d_i \mid d \in (0,1), 0 \le i \le n\text{-}1\}$ is the operation needed to observe the fault effect, where r^d_i means "read the content of the cell i and verify that its value is equal to d".

From AFP1 = (00, w^0_1, 11, 10) and AFP2 = (00, w^1_1, 11, 01) we obtain the following Test Patterns:

- TP1 = (00, w^0_1, r^1_0) , TP2 = (00, w^1_1, r^0_0)

3. Linked Fault: Concept & Modeling

In some cases it is possible that the effect of a FFM influences another functional fault. If these faults share the same aggressor and/or victim cells, the FFMs are called *Linked*, otherwise they are called simple or un-linked and each fault is independent from the others. To understand the concept of linked faults we can consider, as an example, the Disturb Coupling Faults [14] described by the following two FPs:

$$FP_1 = < 0w_1 ; 0 / 1 / - >, \; FP_2 = < 0w_1 ; 1 / 0 / - > \quad (6)$$

A general case is represented in Figure 1, in which a *n* cells memory is affected by two FPs (FP1 and FP2) having different *a*-cells (a_1, a_2) and the same *v*-cell (v). The vertical arrow shows the address order of the memory (from the lowest memory address to the highest) in which *i*, *j* and *k* represent the address of a_1, a_2 and *v*, respectively. By first performing "$0w_1$" (FP1) on cell *i*, the *v*-cell *k* flips from 0 to 1; than performing "$0w_1$" (FP2) on cell *j*, the *v*-cell *k* changes its value again, from 1 to 0. The global result is that the fault effect is masked by the application of FP2, since FP2 has a fault effect (F) opposite to FP1. Looking at the example of Fig. 1, we can derive a rigorous definition of a Linked Fault (LF):

Definition 6: Two FPs, FP1 = <S1/F1/R1> and FP2 = <S2/F2/R2>, are said to be *Linked,* and denoted by "FP1 → FP2", if both of the following conditions are satisfied:

- FP2 masks FP1, i.e., F2 = not (F1);
- The Sensitizing operation (S2) of FP2 is applied after S1, on either the a-cell or v-cell of FP1.

To detect linked faults (LFs), one must detect in isolation (i.e., without allowing the other FP to mask the fault) at least one of the FPs that compose the fault [10]. We extend the concept and the notation described in Definition 6 resorting to AFP (4) formalism.

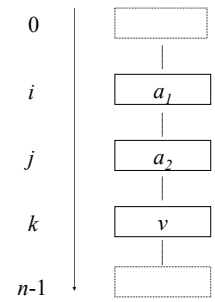

Figure 1 : Example of Linked Fault

Definition 7: Two AFPs, AFP1 = (I_1, E_{s1}, F_{v1}, G_{v1}) and AFP2= (I_2, E_{s2}, F_{v2}, G_{v2}) are said to be *Linked,* and denoted by "AFP1 → AFP2" if:

- $I_2 = F_{v1}$: the state reached by AFP1 is equal to initial state of AFP2;
- AFP2 masks AFP1 : $V(F_{v2})$ = NOT [$V(F_{v1})$] where V(s) function extracts the victim cell from the memory state s

As an example the Linked Fault represented in (6), includes two AFPS obtained by FP1 and FP2:

- AFP1 = (000, w^0_1, 101, 100) (7)
- AFP2 = (101, w^1_1, 110, 111)

If we set the size of memory cell $n = 3$ (0,1,2), the a-cell is cell 0 in AFP1 and cell 1 in AFP2. Both AFP have *v*-cell equal 2.

These two AFP satisfy the constraint of Definition 7:

- $I_2 = 101 = F_{v1}$
- $V(F_{v2}) = 0 = \text{NOT } [V(F_{v1}) = 1]$

Each Linked fault can be modelled by two AFP according to Definition 7. The last step generates the relative Test Patterns (5) able to cover the AFPs, in such a way that from "AFP1 → AFP2" we obtain:

$$\text{"TP1} \rightarrow \text{TP2"} \quad (8)$$

Where

- $TP1 = (I_1, E_1, O_1)$, $TP2 = (I_2, E_2, O_2)$

4. Fault Graph and Memory Model

The generation of a functional test algorithm consists in finding a sequence of memory operations able to initialize the memory in a given state, to sensitize and to observe all the faulty behaviors in the target fault list

For each AFP, the Test Pattern notation (5) already contains this information, hence the generation task become finding a set of TPs able to cover the entire fault list, minimizing the number of operations required. To tackle the problem we resort to the model proposed in [15] to represents the behavior of both the good and the faulty memory. An n one-bit cells memory can be represented as a deterministic Mealy Automata, formally defined as:

$$M = (Q, X, Y, \delta, \lambda) \tag{9}$$

where:

- $Q = \{ (0|\ 1|\ -)^n \}$ is the set of possible *memory states*;
- X is the input alphabet defined in (2);
- $Y = \{0,1,-\}$ is the *output alphabet*, composed of the possible values read as a result of a read operation; '–' denotes the value obtained when a write operation is performed;
- $\delta = Q \times X \rightarrow Q$ is the *state transition function;*
- $\lambda = Q \times X \rightarrow Y$ is the *output function*.

The memory model defined in (8) can be represented as a labeled direct graph

$$G = \{V, E\} \tag{10}$$

where:

- V is the set of vertices, each vertex representing one of the possible states of the memory; $|V| = 2^n$,
- E is the set of edges, each edge representing one of the possible memory operations that cause the transition from a vertex u to a vertex v; the kth label associated with the kth edge has the following representation:

$$Label_k = x / d \tag{11}$$

  where:
  - $x \in X$ is a memory operation
  - $d = \lambda(v,x)$, $d \in Y$ is the output value obtained when performing the operation x when the memory is in the state v.

As an example, Figure 2 shows the model of a 2 bit memory, conventionally named G_0 in the sequel. In G_0, the letters i and j are used to identify the first and the second cell, respectively. Hereinafter, we shall assume $i < j$. According to (8) each Linked Fault is covered by a sequence of two Test Pattern TP1 and TP2, each TP can be modeled on G_0 by a single additional edge (*faulty edge*) [15] The couple of Linked TPs is represented on the graph by adding two extra faulty edges as shown in Figure 3. In the graph representation, the memory state reached by TP1 (F_{v1}) is equal to I_2 (initial state of TP2) in order to satisfy Definition 7.

The graph including the faulty edges is named *Pattern Graph* (PG) and is defined as:

$$PG = \{V_p, E_p \cup F_p\} \tag{11}$$

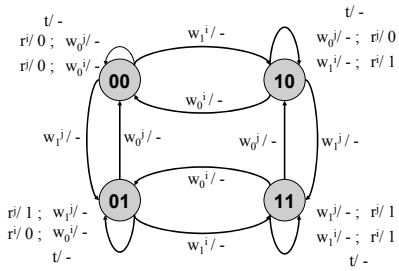

Figure 2: Fault Free Memory Model G_0

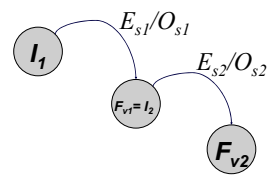

Figure 3: Linked Test Pattern Representation

where each vertex $v \in V_p$ is associated to a memory state, E_p is the set of edges modeling the fault free memory, and F_p is the set of faulty edges. If n is the number of cells composing the target memory the number of nodes composing the pattern graph is $|V_p| = 2^n$. The cardinality of PG vertices is $2^{\max(\#f\text{-}cells)_i}$ with $0 \leq i \leq \#FP$, where #FP represents the number of FPs in the target fault list [15].

Definition 8: given f_l , $f_k \in F_p$, f_l masks f_k if and only if $V(F_{vk}) = V(I_l)$, where f_l is incident from vertex I_l is the and f_k is incident in vertex F_{vk} . V(s) is defined in Definition 7

As an example Disturb Coupling Fault linked to Disturb Coupling Fault [10] is modeled as two FPs:

$$< 0w_1 ; 0 / 1 / - > \rightarrow < 1w_0 ; 1 / 0 / - > \tag{12}$$

Expressing the FPs in terms of AFPs we obtain:

$$(00, w^i_1, 11, 10) \rightarrow (11, w^1_0, 00, 01) \tag{13}$$

The above AFPs are covered by Test Patterns:

$$(00, w^i_1, r^j_0) \rightarrow (11, w^1_0, r^j_1) \tag{14}$$

The PG (named PG_{CF} in the sequel) modeling LF (12) is shown in Figure 4 where the bold edges represent the additional *faulty edges* of (14).

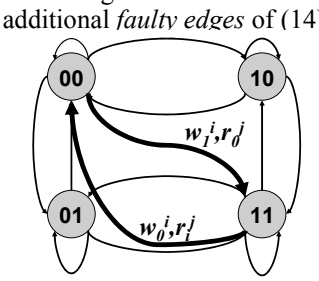

Figure 4: Linked CF Pattern Graph (PG_{CF})

5. March Tests Generation Algorithm

As pointed out in [15], the problem of generating March Tests consists in finding a sequence of TPs able to cover all the memory faults in the target fault list. Since each TP is represented by a faulty edge, the problem is equivalent to finding a sequence of edges (walk) on the pattern graph traversing at least all the faulty edges. The memory operations are easily deduced from the labels of the edges composing the walk.

Definition 9 : *Sequence of Operations* (SO) is a sequence of memory operation SO= $so_1\ so_2$..., with $so_i \in \{X\}$ (see Definition 2). A walk is directly translated into a SO by writing the label of the edges in the sequence.

In the follow we summarize some definitions taken from [15] in order to reduce the complexity of the algorithm.

Definition 10: A *March Test* (MT) is a sequence of March Elements; each *March Element* (ME) is a sequence of memory operations applied on every cell in a specific Address Order AO (increasing, decreasing, random).

The problem is to find a sequence of SOs able to cover all the faults in the fault list while respecting the Definition 9, and without causing masking. The strategy adopted by the algorithm will be to extract a sequence of SOs where each SO will correspond to a March Element.

Definition 11: a *Sequence of Operations* is *valid* if and only if all the operations are performed on the same memory cell *i*; otherwise the sequence causes a constraint violation.

The *valid* SO property ensures that the operations belonging to the SO will be applied on each memory cell as a ME.

Definition 12: The Address Specification of a valid SO is the memory address on which the operations of the SO are performed.

Definition 13: $f_i \in F_p$ is SO Compatible if its address is equal to the address specification of a valid SO and if f_j does not mask $f_j \in$ SO.

The algorithm works on the PG. It attempts to generate a set of March Elements by building *valid* sequences of operations SO. A valid SO can be directly translated into a March Element by specifying its address order and by removing the address specification; considering the G0 memory model (i < j), the march element address order is defined as follow:

- If the address specification of a valid SO is equal to *i* then the address order will be '⇑'
- If the address specification of a valid SO is equal to *j* then the address order will be '⇓'

The algorithm mainly consists in finding a path on the graph able to touch each faulty edge exactly once while respecting the March Test constraints. The main steps of the algorithm are summarized in Fig. 5.

1. **Repeat**
   a. Initialize the Sequence of Operations (SO=∅)
   b. **While** (the next $f \in F_p$ is *SO* Compatible)
      i. Put the *f* into the Sequence of Operations SO
      ii. Delete *f*
   c. **If** (The Sequence of Operations contains at least one *f*) **then**
      i. Apply the Sequence of Operations to each memory cell[1]
      ii. **If** (new *f*s are covered) **then** delete the covered *f*s
      iii. Translate the Sequence of Operations into a March Element by setting its address order
      iv. Print the March Element
   d. **Else**
      i.Report that the *f* cannot be cover by the March Test
3. **Until** (F_p is empty)

Figure 5: March Generation Algorithm

6. Experimental Results

This section reports some experimental results obtained by applying the proposed generation algorithm to different fault lists. The algorithm has been implemented in about 900 lines of C++ code, compiled with gcc compiler. All the experiments are performed on a *ASUS, AMD 1500Mhz* based Laptop with 512 MB of RAM. We performed two classes of experiments, targeting two different fault lists. The fault lists include the set of realistic linked faults presented in [10] :

1. Fault List #1 includes single, two and three cells LFs [10]
2. Fault List #2 includes the single cell LFs [10].

All generated March Tests have been verified using an ad hoc memory fault simulator [13] able to validate their correctness w.r.t. the target Fault list.

We compare our new generated Test Algorithms with the previous march tests targeting the same fault list. In particular we consider:

- **43*n* March Test** : it is the only march test targeting linked faults automatically generated. It is able to deal only with a reduced subset of Fault List #1. Published in [11]
- **41*n* March SL**: it is the state of the art in terms of complexity and coverage. It is generated by hand; it covers Fault List #1. Published in [10].
- **11*n* March LF1**: it is the well known march test able to cover Fault List #2 Published in [16].

[1] E.g., if the address of the Sequence of Operations was *i*, now we try to apply the same sequence to the cell *j* and so on …

Table 1 reports March Tests generated for different fault lists, it also shows the generation time required by the algorithm in terms of CPU time (in seconds). We generated unpublished March Test reducing the complexity of the previous ones. We obtained two March Tests targeting fault list #1, March ABL and March RABL, respectively 37*n* and 35*n* of complexity. Comparison results (column 5) show a reduction of test length, of **13.9%** w.r.t 43*n* March Test, and **9.7%** w.r.t 41*n* March SL. Finally 9*n* March ABL1, targeting fault list #2, reduces test length of **18.1%** w.r.t 11*n* March LF1.

7. Conclusions

This paper presented a methodology to automatically generate March Tests. A general model has been used to represent both known memory faults, and to possibly add new user-defined faults. In particular we address Static Linked Faults. With respect to previously presented approaches our methodology allows generating non-redundant March Tests in a very low computation time, and without exhaustive searches. We have been able to generate March Tests for the complete set of Static Linked Faults obtaining new test algorithms. Comparison with state of the art march tests show that we reduce the complexity of the well known tests and therefore we reduce the time to test making our solution very attractive from the industrial point of view. On going activities are focused on the extension of the model to multi-port memory linked faults and on the possibility of introducing additional constraints on the generated March Test. In particular, has been demonstrated in literature that March Tests with particular address orders (i.e., all increasing or all decreasing) can be implemented more efficiently, we want to be able to add to our model these constraints.

References

[1] International Technology Roadmap for Semiconductors, *"International technology roadmap for semiconductors 2004 Update"*, http://public.itrs.net/Home.htm, 2004

[2] R. Dekker, F. Beenker, L. Thijssen, *"*A Realistic Fault Model and Test Algorithms for Satic Random Acces Memory*"*, IEEE Transaction on Computer-Aided Design, Volume: 9, Issue: 6, June 1990

[3] R.D. Adams and E.S. Cooley, *"Analysis of a Deceptive Destructive Read Memory fault Model and Recommended Testing",* NATW 1996. 5th IEEE North Atlantic Test Workshop, 1996

[4] Z. Al-Ars, Ad J. van de Goor, "S*tatic and Dynamic Behavior of Memory Cell Array Opens and* Shorts *in Embedded DRAMs*", DATE 2001, IEEE Design Automation and Test in Europe, 2001, pp. 496-503.

[5] Z. Al-Ars and A.J. van de Goor, "*Approximating Infinite Dynamic Behavior for DRAM Cell Defects*", VTS 2002, 20th IEEE VLSI Test Symposium, 2002, pp.401-406.

[6] D. S. Suk, S. M. Reddy, *"A March Test for Functional Faults in Semiconductor Random-Access Memory"* IEEE Transaction on Computer-Aided Design, Volume: 30, Issue: 12, 1981

[7] A. J. van de Goor, G.N. Gayadadjiev, V.N. Yarmolik, V.G. Mikitjuk, *"March LA: A Test for Linked Memory Faults",* ED&TC 1997, Proc. European Design and Test Conference, 1997, pp. 167

[8] A. J. van de Goor, G.N. Gayadadjiev, V.N. Yarmolik, V.G. Mikitjuk, *"March LR: A Test for Realistic Linked Faults"*, VTS 1996, 16th IEEE VLSI Test Symposium, 1996, pp. 272-280.

[9] S. Hamdioui, Z. Al-Ars, A.J. van de Goor, M. Rodgers, "*March SL: a test for all static linked memory faults"*, ATS 2003, 12th IEEE Asian Test Symposium, 2003. pp. 372 – 377

[10] S. Hamdioui, Z. Al-Ars, A. J. van de Goor, M. Rodgers, *"Linked Faults in Random Access Memories Concept Fault Models Test Algorithms and Industrial Results"*, IEEE Transaction on Computer-Aided Design, Volume: 23, Issue: 5, May 2004, pp. 737-757

[11] S. M. Al-Harbi, S. K. Gupta*, "Generating Complete and Optimal March Tests for Linked Faults in Memories",* VTS 2003, 21th IEEE VLSI Test Symposium, 2003, pp. 254 -261

[12] A. J. van de Goor, Z. Al-Ars, *"Functional Memory Faults: A Formal Notation and a Taxonomy"*, VTS 2000, 18th IEEE VLSI Test Symposium, 2000, pp. 281-289.

[13] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, *"Specification and design of a new memory fault simulator"*, ATS 2002, 11th IEEE Asian Test Symposium, 2002.pp. 92 – 97.

[14] A. J. van de Goor, B. Smit, "*Generating March Tests Automatically*", ITC 1994, IEEE International Test Conference, 1994, pp.870-877, 1994

[15] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto, "*Automatic March Tests Generation for Static and Dynamic Faults in SRAMs*", ETS 2005, 10th IEEE European Test Symposium, 2005.

[16] S. Hamdioui, Z. Al-Ars, A. J. van de Goor, *"A Fault Primitive Based Analysis of Linked Faults in RAMs"*, MTDT 2003, IEEE International Workshop on Memory Technology, Design and Testing, 2004,

March Test	Algorithm	Fault List	CPU Time (s)	O(n)	Improve (%)		
					43*n* March Test	41*n* March SL	11*n* March LF1
ABL	$\Updownarrow(w_0)$ $\Uparrow(r_0,r_0,w_0,r_0,w_1,w_1,r_1)$ $\Uparrow(r_1,r_1,w_1,r_1,w_0,w_0,r_0)$ $\Downarrow(r_0,w_1)$ $\Downarrow(r_1,w_0)$ $\Downarrow(r_0,r_0,w_0,r_0,w_1,w_1,r_1)$ $\Downarrow(r_1,r_1,w_1,r_1,w_0,w_0,r_0)$ $\Uparrow(r_0,w_1)$ $\Uparrow(r_1,w_0)$	**#1**	**1.03**	**37*n***	**13.9%**	**9.7%**	**-**
RABL	$\Updownarrow(w_0)$ $\Uparrow(r_0,r_0,w_0,r_0)$ $\Uparrow(r_0,w_1,r_1,r_1,w_1,r_1,w_0,r_0)$ $\Uparrow(r_0,w_1)$ $\Downarrow(r_1,r_1,w_1,r_1,w_0,r_0,w_0,r_0)$ $\Uparrow(w_1)$ $\Uparrow(r_1,r_1,w_1,r_1,w_0,r_0,r_0,w_0,r_0,w_1,r_1)$	**#1**	**1.35**	**35*n***	**18.6%**	**14.6%**	**-**
ABL1	$\Updownarrow(w_0,)$ $\Updownarrow(w_0,r_0,r_0,w_1)$ $\Updownarrow(w_1,r_1,r_1,w_0)$	**#2**	**0.98**	**9*n***	**-**	**-**	**18.1%**

Table 1: Experimental Results