Linear Test Sequences for Detecting Functionally Faulty RAM's

Guy Even * Ophir Rachman * Ilan Spillinger †

Abstract

In this paper we deal with the detection of functional faults in RAM's. We present a fault model which consists of decoder faults and memory array faults. We rectify and extend previous definitions of decoder faults, and we refine previous definitions of two coupling faults. Three test sequences of length $9N$, $13N$ and $17N$ are presented with proofs of increasing detection capabilities. Our test sequence of length $17N$ is capable of detecting all combinations of faults in our model.

keywords: random access memories (RAM's), functional faults, fault models, fault detection, test sequences.

*Computer Science Dept., Technion, Haifa, Israel.
†IBM T.J. Watson Research Center, Yorktown Heights, NY.

1 Introduction

Rapid developments in semiconductor technology have resulted in continuing growth of larger and denser random access memories (RAM's) on a single chip. RAM's are customarily available in sizes of N words and m bits per word (in this paper we deal with $m = 1$). Many efforts for designing test procedures to detect different failure models for RAM's have been done in the past [1, 2, 3, 4, 5, 6, 7, 8].

In this work we deal with functional testing of RAM's. Such tests proceed by applying a sequence of read and write operations to the RAM, and by verifying that the values returned by the read operations are as expected.

Our goal is to design linear complexity test sequences that cover a wide variety of functional faults. In order to achieve this goal, we concentrate on the following two objectives: a) Defining an accurate fault model. b) Designing test sequences that detect simultaneous occurrence of all functional faults in the model. These objectives have not been fulfilled previously.

The first objective is achieved by presenting a fault model based on ideas used in [4, 6, 8]. The main faults presented in these works are stuck-at, coupling, and multiple access faults. The first two types are caused by defects in the memory cells of the RAM, and the third is caused by defects in the decoder of the RAM. In our model, we refine the definition of coupling faults, by distinguishing between two types of coupling faults. This refinement was previously considered in the context of pattern-sensitive faults (see [1]). In addition, we characterize decoder faults in an accurate way, which rectifies previous definitions of [1, 3, 6, 8]. It is important to note, that our modified fault model includes the faults described in [3, 6, 7, 8] except for transition faults.

The second objective is achieved by introducing a test sequence of length $17N$, which detects all combinations of faults of our fault model. In addition, we introduce shorter test sequences of length $9N$ and $13N$ which detect combinations of faults that belong to subsets of our fault model. Previous linear test sequences either detect weaker fault models [6], or have not been proven to detect simultaneous occurrences of all the faults of the fault model [8].

This paper is organized as follows. In Section 2, we define the fault model. In Section 3, we describe the test sequences introduced in this work. In section 4, we prove the detection capabilities of our test sequences. In section 5, we overview the main contributions of our work, and compare them with the literature. We postpone this section due to the need to use terminology and definitions presented in section 2. In section 6, we summarize this work, and discuss a few issues related to possible modifications of the sequences presented in this work.

2 The fault model

In this section we describe our fault model. The fault model is derived by considering possible faults in each of the RAM's sub-components. Our analysis is characterized by differentiating between memory addresses and memory cells. Addresses are denoted by numbers in the range $[1, .., N]$, and memory cells are denoted by $[c_1, ..c_N]$.

In general, functional test sequences apply read and write operations to the tested RAM. A $read(i)$ operation is issued with an address i as input, and this operation is expected to return the state of memory cell c_i. In a $write(i, b)$ operation, an address i and a bit b are input to the RAM, and and it is expected that b will be the state of memory cell c_i after the $write(i, b)$ operation is issued.

2.1 Decoder Faults

In this section we deal with decoder faults. We define functionally faulty decoders and explain the effects of a faulty decoder to read and write operations.

We assume that decoders, faulty or not, behave deterministically and consistently (i.e., decoders are non-sequential). Therefore, we can represent a decoder as a function $access : \{1, .., N\} \longrightarrow \{A : A \subseteq \{c_1, .., c_N\}\}$. The set $access(i)$ is the set of memory cells accessed when the decoder is given the address i. A correct decoder is designed so that its access function satisfies $access(i) = \{c_i\}$ for every address i.

The effect of $read(i)$ and $write(i, b)$ operations depend on whether the set $access(i)$ is empty or not:

$access(i)$ **is empty :** If the set $access(i)$ is empty, a $write(i, b)$ operation has no effect on the states of the memory cells. The outcome of a $read(i)$ operation depends on the design and the technology of the RAM (see [3] for a full discussion).

We consider two options: In the first option, the value returned is constant. In the second option, the value returned is equal to the last value returned by a read operation issued before the read operation $read(i)$. (If a $read(i)$ operation is the first read operation, then it returns a random value).

In the first option we regard memory cell c_i to be stuck-at (this is a memory array fault defined in section 2.2). Since the first option is covered by a memory-array fault, we assume that the outcome of a $read(i)$ operation is determined according to the second option.

$access(i)$ **is non-empty :** When the set $access(i)$ is non-empty, a $write(i, b)$ operation writes the value b to all the memory cells of the set $access(i)$.

The outcome of a $read(i)$ operation, when the set $access(i)$ is not empty, is determined by the states of the accessed cells. Formally, we attach to each address i for which the set $access(i)$ is not empty, a boolean function $R_i : \{0, 1\}^{|access(i)|} \longrightarrow \{0, 1\}$ called the extended-read function of address i. The extended-read function of address i determines the outcome of a $read(i)$ operation as following: let $access(i) = \{c_{i_1}, .., c_{i_k}\}$, and let b_j denote the state of cell c_{i_j}. The value returned by a $read(i)$ operation is defined to be $R_i(b_1, .., b_k)$. (In particular, the extended-read functions can be the boolean AND or OR functions.) Note, that although we do not restrict the extended-read functions, they are fixed (for every address i), and do not change during the execution of the test sequence.

We define a decoder to be functionally incorrect if it has one of the following faults:

stuck-open faults : There exists an address i, such that the set *access*(i) is empty.

collision faults : There exist two different addresses i and j, such that the intersection of the sets *access*(i) and *access*(j) is non empty.

A collision fault, $access(i) \cap access(j) \neq \phi$, is *activated* whenever a write or a read operation is issued with the addresses i or j.

2.2 Memory Array Faults

The memory array contains memory cells, each memory cell can be in one of two states, 0 or 1.

We consider two types of memory array faults: stuck-at faults and coupling faults. The difference between previous models and our model is in the definition of coupling faults. We divide coupling-faults into two classes: transition-coupling faults and write-coupling faults.

We use a definition introduced by [6] of a *forced-write*. The event $forced\_write(c_i, b)$ happens whenever a $write(j, b)$ operation is issued with an address j which satisfies $c_i \in access(j)$. We denote a forced-write in which the state of cell c_i is *changed* from 0 to 1 by $forced\_write(c_i, \uparrow)$. Similarly, $forced\_write(c_i, \downarrow)$ denotes a forced-write in which the state of cell c_i is *changed* from 1 to 0.

stuck-at faults : Memory cell c_i is said to be stuck-at state 0 (1), if its state is always 0 (1).

transition-coupling faults : Let c_i and c_j be two memory cells, and $b \in \{0, 1\}$. Cell c_j is defined to be *transition-coupled* with state b to the transition $\uparrow$ of cell c_i, if the event $forced\_write(c_i, \uparrow)$ causes cell c_j to be in state b. We denote this fault by $forced\_write(c_i, \uparrow) \Rightarrow (c_j = b)$.

Similarly, cell c_j is defined to be *transition-coupled* with state b to the transition $\downarrow$ of cell c_i, if the event $forced\_write(c_i, \downarrow)$ causes cell c_j to be in state b. We denote this fault by $forced\_write(c_i, \downarrow) \Rightarrow (c_j = b)$.

write-coupling faults : Let c_i and c_j be two memory cells, and $b_1, b_2 \in \{0, 1\}$. Cell c_j is defined to be *write-coupled* with state b_2 to the event $forced\_write(c_i, b_1)$, if this event causes cell c_j to be in state b_2. We denote this fault by $forced\_write(c_i, b_1) \Rightarrow (c_j = b_2)$.

Remarks:

$-$The difference between write-coupling faults and transition-coupling faults lies in the operations which activate the fault. A write-coupling fault, $forced\_write(c_i, b_1) \Rightarrow (c_j = b_2)$, is activated by any forced-write operation that writes the value b_1 to cell c_i, regardless of the previous state of c_i. On the other hand, a transition-coupling fault, $forced\_write(c_i \downarrow) \Rightarrow (c_j = b_2)$, is only activated by forced-write operations which *change* the state of cell c_i from state 1 to state 0. (The distinction between transition-coupling faults and write-coupling faults was previously made in the context of pattern-sensitive faults, see [1]).

−There is no transitivity of activated faults. Since coupling-faults are activated only by *forced-write* operations, a change in the state of memory cells caused by an activated coupling-fault does not activate other coupling-faults.

−In previous works, decoder faults have been reduced to memory array faults [3, 4, 6, 7]. However, this reduction is inaccurate, and it fails even for modeling the effect of decoder faults on write operations [8]. For example, consider a RAM with four memory cells ($N = 4$) that has the following decoder faults (and no other faults): $access(1) = \{c_1\}$, $access(2) = \{c_2\}$, $access(3) = \{c_1, c_2\}$ and $access(4) = \{c_3, c_4\}$. Suppose one tries to model this faulty decoder using memory array faults. Since $c_1, c_2 \in access(3)$, one should add the fault $forced\_write(c_1, b) \Rightarrow (c_2 = b)$ or the fault $forced\_write(c_2, b) \Rightarrow (c_1 = b)$. Adding either of these faults will not properly model the effect of $write(c_1, b)$ and $write(c_2, b)$ operations. Therefore, in this RAM, faults associated with write operations cannot be modeled by write-coupling faults. On the other hand, faults associated with write operations can be modeled by write-coupling faults for cases in which distinct representatives exist.

−Our modified fault model does not include transition-faults [3, 4, 7, 8]. In this fault a memory cell fails to undergo a transition from state 0 to state 1 or from state 1 to state 0. Instances of transition faults which do not appear simultaneously with write-coupling faults or with transition-coupling faults are easily reducible to stuck-at faults, and indeed, all of our test sequences can detect such instances of transition-faults. However, it is an open question whether our test sequences are capable of detecting all combinations of transition-faults, write-coupling faults, and transition-coupling faults (the same holds for other linear test sequences such as [8]).

2.3 Faults in Other Components

In this section we discuss faults in the address register, data registers and read/write logic (see [1] for a description of these components). Following [5] (with slight differences) we show how these faults can be reduced to memory array faults and decoder faults. We consider only stuck-at bits in these registers, since these registers are designed separately from the memory array.

2.3.1 the address register

Stuck-at bits in the address register can be reduced to decoder faults (i.e. collision faults).

2.3.2 the data register

We deal with single-bit memory cells, and therefore a stuck-at data register can be reduced to a RAM in which all memory cells are stuck-at. (This reduction still holds when there are separate registers for the read and write values).

2.3.3 read/write logic

Stuck-at faults in the read/write logic cause either constant read values or constant write values. These faults are reduced to stuck-at faults in all the memory cells of the RAM.

2.4 Resolving Conflicting Faults

In the fault model presented so far it is possible for different faults to have conflicting effects. For example, two coupling faults might want to cause the same memory cell to be in different states. We resolve conflicting interactions of faults by specifying the following rules:

1. The state of a stuck-at cell cannot be modified at all.
2. A $write(i, b)$ operation causes the state of all non-stuck-at cells in the set $access(i)$ to be equal to b, regardless of any coupling-faults.
3. We do not specify how conflicting coupling-faults interact. However, we assume that the same coupling-fault dominates whenever the same coupling-faults are activated simultaneously.

If a RAM has a conflicting combinations of faults, then we only consider the subset of faults which can be activated. The subset of faults which can not be activated is omitted because it does not affect the functional behavior of the RAM.

3 The test sequences

In this section we describe the test sequences T_{9N}, T_{13N} and T_{17N}. We describe these sequences for even values of N.

We use the following notation in the description of these sequences in figures 1 - 3. The order of the operations is from left to right. The row of the operation determines the address accessed. A 'R' denotes a read operation. A '0' ('1') denotes a write operation which writes a 0 (1).

Address	Init	part 1	part 2	part 3	part 4
1	1	$R0$	$R0$	$R1$	$R1$
2	0	$R1$	$R1$	$R0$	$R0$
3	1	$R0$	$R0$	$R1$	$R1$
$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\cdots$	$\cdots$
N	0	$R1$	$R1$	$R0$	$R0$

Figure 1: Sequence T_{9N}

Address	Init	part 1	part 2	part 3	part 4
1	1	$R10$	$R10$	$R01$	$R01$
2	0	$R01$	$R01$	$R10$	$R10$
3	1	$R10$	$R10$	$R01$	$R01$
$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\cdots$	$\cdots$
N	0	$R01$	$R01$	$R10$	$R10$

Figure 2: Sequence T_{13N}

Address	Init	part 1	part 2
1	1	$R010$	$R010$
2	0	$R101$	$R101$
3	1	$R010$	$R010$
$\vdots$	$\ddots$	$\ddots$	$\ddots$
N	0	$R101$	$R101$

Address	part 3	part 4
1	$R101$	$R101$
2	$R010$	$R010$
3	$R101$	$R101$
$\vdots$	$\cdots$	$\cdots$
N	$R010$	$R010$

Figure 3: Sequence T_{17N}

4 The detection capabilities of our test sequences

In this section we prove the detection capabilities of the test sequences T_{9N}, T_{13N} and T_{17N}. We prove the following:

Theorem 1: Sequence T_{9N} detects all combinations of stuck-at faults and decoder faults

Theorem 2: Sequence T_{13N} detects all combinations of stuck-at faults, decoder faults and write-coupling faults.

Theorem 3: Sequence T_{17N} detects all combinations of stuck-at faults, decoder faults, write-coupling faults and transition-coupling faults.

For the proofs of the above theorems, we first state four lemmata. The first lemma characterizes access functions which are permutations.

Lemma 1: Let $access : [1,..,N] \to \{A : A \subseteq [c_1,..,c_N]\}$ be a function. The function *access* satisfies the following three conditions:

(1) For every i the set $access(i)$ is not empty. Formally,

$$\forall i \quad : \quad access(i) \neq \phi$$

(2) For every j the set $access(j)$ is not included by the union of the sets $access(1),..,access(j-1)$. Formally,

$$\forall j \quad : \quad access(j) \not\subseteq \bigcup_{i<j} access(i)$$

(3) For every j the set $access(j)$ is not included by the union of the sets $access(j+1),..,access(N)$. Formally,

$$\forall j \quad : \quad access(j) \not\subseteq \bigcup_{i>j} access(i)$$

if and only if the function *access* is a permutation (i.e. *access* is a one-to-one function in which for every $i : |access(i)| = 1$).

The proof of lemma 1 is given in the appendix.

The following three lemmata point out some important properties of our sequences. These properties follow directly from the definition of the sequences.

Lemma 2: The sequence T_{9N} satisfies the following property: The write operations of part 2 are an exact repetition of the write operations of part 1. In particular, all the collision faults which are activated during part 1 are also activated during part 2, and vice-versa. Moreover, these faults are activated in the same order in parts 1 and 2. (The same property holds for parts 3 and 4).

Lemma 3: The sequence T_{13N} satisfies the following property: The write operations of part 2 are an exact repetition of the write operations of part 1. In particular, all the write-coupling and collision faults which are activated during part 1 are also activated during part 2, and vice-versa. Moreover, these faults are activated in the same order in parts 1 and 2. (The same property holds for parts 3 and 4).

Lemma 4: The sequences T_{9N}, T_{13N} and T_{17N} satisfy the following property: For every address i, the $read(i)$ operations in parts 1 and 2 are expected to return complementary values. (The same property holds for parts 3 and 4).

We now prove theorems 1-3.

Proof of Theorem 1: Let M be a faulty RAM which has a combination of stuck-at, stuck-open, and collision faults. We consider three cases:

case I : M has only stuck-at faults.

case II : M has stuck-open faults (and maybe additional types of faults).

case III : M has collision faults but it lacks stuck-open faults (it may also have stuck-at faults).

Case I: If only stuck-at faults are present, then there exists a memory cell c_i that is always in the same state. Since there are no decoder faults, $read(i)$ operations return the state of cell c_i. However, since during the test sequence T_{9N} both 0 and 1 are expected to be read from each address, it is certain that the sequence T_{9N} detects M as faulty.

Case II: If memory M has stuck-open faults, let i be an address with an empty $access(i)$ set. Assume towards a contradiction, that the test sequence T_{9N} does not detect memory M as faulty, that is, all read operations return the expected result. We consider two cases:

(a) $i > 1$. In this case the $read(i-1)$ operation in part 1 returns the expected value, say b. Since address i is stuck-open, the $read(i)$ operation in part 1 also returns the value b. Since b is *not* the expected return value (recall that two consecutive read operations are expected to return complementary values), the test sequence detects memory M as faulty, a contradiction.

(b) $i = 1$. One can prove this case using part 3 of the test sequence applying arguments similar to the ones used in case (a).

case III: Suppose M does have collision faults but lacks stuck-open faults. We start by showing that if sequence T_{9N} does not detect a fault in M, then the following condition can not be satisfied:

$$\text{Condition (A)} \qquad \exists j \;:\; access(j) \subseteq \bigcup_{i<j} access(i)$$

According to lemma 4, the $read(j)$ operations in parts 1 and 2 return complementary values. Therefore, there exists at least one cell, $c_k \in access(j)$, having different states during the two $read(j)$ operations. We show that this is impossible if condition (A) holds.

Consider the prefix of part 1 up to the $read(j)$ operation, and the same prefix of part 2. According to lemma 2, the same collision faults are activated in these prefixes, and the order in which these faults are activated is the same as well. If condition (A) holds, there exists $i < j$ such that $c_k \in access(i)$, and therefore, at least one fault which writes to cell c_k is activated during both of the prefixes. Moreover, the states of cell c_k, when the $read(j)$ operations are issued in parts 1 and 2, are determined by the last fault that writes to cell c_k during these prefixes. This implies that the states of cell c_k during the two $read(j)$ operations in parts 1 and 2 are identical, a contradiction.

Let condition (A') be the following modification of condition (A):

$$\text{Condition (A')} \qquad \exists j \;:\; access(j) \subseteq \bigcup_{i>j} access(i)$$

By applying the same arguments on parts 3 and 4 of the test sequence T_{9N}, we can show that if sequence T_{9N} does not detect a fault in M, then condition (A') can not hold.

The analysis of case III up to this point implies: if memory M has collision faults but lacks stuck-open faults, and if sequence T_{9N} does not detect faults in memory M, then memory M must satisfy the following three conditions:

1. $\forall i \;:\; access(i) \neq \phi$.
2. $\forall j \;:\; access(j) \not\subseteq \bigcup_{i<j} access(i)$.
3. $\forall j \;:\; access(j) \not\subseteq \bigcup_{i>j} access(i)$.

According to lemma 1, satisfying these three conditions implies that *access* is a permutation. This implies that memory M does *not* have decoder faults, a contradiction to the assumption that case III holds.

Cases I,II and III cover all possible faulty RAM's having a combination of stuck-at faults, stuck-open faults and decoder faults. Since we proved that in all of these cases sequence T_{9N} detects a fault, the theorem follows.
$\square$

Proof of Theorem 2: Let M be a faulty RAM which has a combination of stuck-at, stuck-open, collision faults and write-coupling faults. We consider four cases:

case I : M has only stuck-at faults.

case II : M has stuck-open faults (and maybe additional types of faults).

case III : M has write-coupling faults but lacks stuck-open faults and collision faults (it may have stuck-at faults).

case IV : M has collision faults but lacks stuck-open faults (it may also have stuck-at faults and write-coupling faults).

The proof that sequence T_{13N} detects M as faulty is identical to the proof that sequences T_{9N} detects M as faulty for cases I and II.

Case III: If write-coupling faults are present, then there exists a fault $\mathit{forced\_write}(c_i, b_1) \Rightarrow (c_j = b_2)$ in memory M. Suppose that $i < j$. If the sequence T_{13N} does not detect M as faulty, then according to lemma 4, the $read(j)$ operations return complementary values during parts 1 and 2. We show that this is impossible.

Consider the prefix of part 1 up to the $read(j)$ operation, and the same prefix of part 2. According to lemma 3, the same write-coupling faults are activated in these prefixes, and the order in which these faults are activated is the same as well. Since $i < j$, at least one write-coupling fault which writes to cell c_j is activated during both of these prefixes. Therefore, the states of cell c_j, when the $read(j)$ operations are issued in parts 1 and 2, are determined by the last write-coupling fault that writes to cell c_j during the prefixes. Since no decoder faults are present, $read(j)$ operations return the state of cell c_j. This implies that the $read(j)$ operations in parts 1 and 2 return the same value, and hence, a contradiction.

The proof runs along the same lines when $i > j$, but uses parts 3 and 4 of the sequence T_{13N}. This proves the theorem when case III holds.

Case IV: Proving that sequence T_{13N} detects a fault in this case is almost identical to the proof of case III in the proof of theorem 1. The only difference is that the state of memory cells might be changed also by write-coupling faults. However, if we use lemma 3 instead of lemma 2, then the same arguments prove that sequence T_{13N} detects M as faulty when case IV holds.

These four cases cover all possible faulty RAM's having a combination of stuck-at faults, stuck-open faults, collision faults and write-coupling faults. Since we proved that in all these cases sequence T_{13N} detects a faults, the theorem follows.
□

Proof Sketch of Theorem 3: The proof of this theorem is very similar to the proof of theorem 2. Lemmata 3 and 4 hold also for sequence T_{17N}. However, it is not true that all the transition-coupling faults which are activated during part 1 of sequence T_{17N}, are also activated during part 2 (and vice-versa). Therefore, the naive modification of lemma 3 to also deal with activated transition-coupling faults does not hold.

This difficulty can be overcome by the following observation about faults activated during sequence T_{17N}:

Let $c_k \in access(j)$ be a memory cell which is affected by at least one fault that is activated before the $read(j)$ operation in part 1 (2). If f is the *last* such activated fault in part 1 (2), then f is also the *last* fault that affects cell c_k which is activated before the $read(j)$ operation in part 2 (1). (The same observation holds for parts 3 and 4).

The proof now follows by using the proof of theorem 2, substituting lemma 3 with the above observation, and substituting write-coupling faults with both types of coupling faults.
□

5 Overview of main contributions

In the previous sections we introduced our fault model, presented three test sequences and proved their detection capabilities. In this section we highlight and compare our work with the literature. We omit it from the introduction section since at this point the reader should be more familiar with the terminology and the definitions needed for explaining the advantages of our fault model and the enhanced detection capabilities of our test sequences. In the following we point out the main contributions of this paper.

1. We give a precise definition of functionally faulty decoders: A non-sequential decoder is functionally faulty if and only if it has stuck-open faults or collision faults. Previous definitions of faulty decoders [1, 3, 6] are inaccurate and include functionally correct decoders. More specifically, these previous definitions define a decoder to be faulty if it accesses wrong memory cells. However, if the correspondence between memory addresses and memory cells is one-to-one and onto, then a decoder is functionally correct, even if for some addresses wrong memory cells are accessed. The work of Nair [5] contains a definition which is similar to ours, however, this definition was ignored in later works.

2. We use a simple and convenient fault model for decoders. Lemma 1 is utilized for proving that our test sequences detect decoder faults. Van De Goor et al. [4] present sufficient conditions for detecting faulty decoders with march tests. However, these conditions are not applicable to our test sequences because our test sequences do not consist of marches (we write alternating values to the RAM).

3. We do not use the common reduction of decoder faults to memory array faults [1, 3, 6, 7]. In the work of Suk and Reddy [8] this reduction is shown to be inaccurate due to the effects of decoder faults on read operations. We show that this reduction even fails for write operations.

4. Our test sequences can detect decoder faults for all extended-read functions. Previous works [5, 8] restricted the extended-read functions to be either always the boolean function OR or always the boolean function AND. Van De Goor et al. [4] considered arbitrary extended-read functions. Although they claim to state conditions which are necessary and sufficient for detecting such faults, they retreat from this claim in the proof of the sufficiency of the conditions. Therefore, they do not provide test sequences capable of detecting decoder faults for all extended-read functions.

5. We define a hierarchy of faults which is used for resolving conflicting combinations of faults. A discussion of conflicting fault combinations appears in [4], this discussion deals separately with various combinations of faults. Our approach is based on the assumption that the RAM behaves deterministically, and therefore, the resulting description of the hierarchy is compact and well defined.

6. We differentiate between two kinds of coupling faults: write-coupling faults and transition-coupling faults (this distinction is also known as static coupling faults and active coupling faults). We show that detecting write-coupling faults is easier than

detecting transition-coupling faults. In fact, we introduce a march test of length $13N$ for detecting write-coupling faults. Suk and Reddy [8] showed a lower bound of $14N$ on the length of march tests which detect transition-coupling faults .

7. We introduce a test sequence of length $17N$ capable of detecting all combinations of decoder faults and memory array faults. Suk and Reddy [8] introduced a sequence of length $15N$ capable of detecting the following combinations of faults: Either all combinations of memory array faults, or all combinations of stuck-at faults and decoder faults. On the other hand, they also deal with another fault called a 'transition-fault' which we did not deal with at all. In the work of Van De Goor et al. [4], decoder faults are dealt separately from memory faults.

8. We introduce a methodology for developing test sequences. We use the same methodology for developing test sequences for three fault models.

6 Summary and discussion

In this section we summarize the main issues presented in the paper, and we suggest several possible directions for further research.

The goals achieved in this paper are: a) defining an accurate fault model which contains ideas of previous attempts to define fault models, and b) designing linear test sequences of length $13N$ and $17N$, which detect wider varieties of faults than previously known test sequences. We achieve these improved detection capabilities by two major techniques.

The first technique proceeds by devising a sequence which is executed twice. This repeated sequence has the following two properties: a) Memory cells that are *not* affected by any faults have *complementary* states when they are accessed in the two executions of the sequence. b) Memory cells that are affected by at least one fault have *identical* states when they are accessed in the two executions of the sequence. This technique enables us to detect faults in the RAM.

The second technique we use, is initializing the memory cells so that even cells are initialized with 1's, and odd cells are initialized with 0's. This initialization enables our sequences to detect stuck-open faults in addition to all other types of faults.

It is interesting to note, that test sequences T_{13N} and T_{17N} can be modified into test sequences of length $12N$ and $16N$, that cover the same fault models, except for stuck-open faults. This can be achieved by following a technique presented in [8], in which a test sequence can be applied to the RAM with no initialization. However, this modification is not recommended for the following two reasons. First, as reported in [3], stuck-open faults are not rare, and occur even more frequently than collision faults. Second, the technique presented in [8] requires linear "space-complexity", that is, when testing a RAM of size N, one must use an auxiliary memory of size N.

Another interesting subject is the difference between write-coupling faults and transition-coupling faults. Detecting write-coupling faults is easier than detecting transition-coupling faults. In particular, previous test sequences that detect transition-coupling faults, also detect write-coupling faults (For example [6, 8]). On the other hand, the test sequence T_{13N} that detects write-coupling faults, is shorter than the lower bound of $14N$ for detecting transition-coupling faults as presented in [8]. Therefore, if write-coupling faults are adequate for modeling coupling faults, then one can do with shorter test sequences.

As for further research, we suggest two directions.

The first direction deals with the word size of the RAM. In this paper we considered only word size $m = 1$, which is not general. It might seem that testing a RAM with word size $m > 1$ can be simply done by treating it as if it is a RAM of size $(N \cdot m)$ with with single bit words. However, it is not clear whether using the fault model which is designed for word size $m = 1$ results in a faithful model for $m > 1$. The problem is that in a RAM with word size $m > 1$ each read/write operation accesses m bits simultaneously, and it is not possible to access a single bit of a word without accessing the rest of the bits of during the same operation.

The second possible direction for further research considers the type of the test sequences. All deterministic test sequences presented upto date are march sequences. It is not clear whether march sequences are indeed optimal for testing RAM's, and it will be very interesting to present shorter test sequences for similar fault models that do not consist of marches.

Acknowledgments

The authors would like to thank Shimon Even for his remarks. Guy Even wishes to thank Benny Chor for his encouragement and support of this work.

References

[1] Abadir, M.S. and H.K. Reghbati, Functional testing of semiconductor random access memories, *ACM Computing Surveys* **15** (3) (1983) 175-198.

[2] David, R., A. Fuentes and B. Courtois, Random pattern testing versus deterministic testing of RAM's, *IEEE Trans. Comput.* **C-38** (5) (1989) 637-650.

[3] Dekker, R., F. Beenker and L. Thijssen, Fault modeling and test algorithm development for static random access memories, *Proc. 1988 IEEE International Test Conference,* Paper 20.1, 342-352.

[4] Van De Goor, A.J. and C.A. Verruijt, An overview of deterministic functional RAM chip testing, *ACM Computing Surveys* **22** (1) (1990) 5-33.

[5] Nair, R., Comments on an optimal algorithm for testing semiconductor random access memories, *IEEE Trans. Comput.* **C-28** (3) (1979) 258-261.

[6] Nair, R., S.M. Thatte and J.A Abraham, Efficient algorithms for testing semiconductor random access memories, *IEEE Trans. Comput.* **C-27** (6) (1978) 572-576.

[7] Papachristou, C.A. and N. B. Sahgal, An improved method for detecting functional faults in semiconductor random access memories, *IEEE Trans. Comput.* **C-34** (2) (1985) 110-116.

[8] Suk, D.S. and S.M. Reddy, A march test for functional faults in semiconductor random access memories, *IEEE Trans. Comput.* **C-30** (12) (1981) 982-985.

Appendix

Lemma 1: Let $access : [1, .., N] \rightarrow \{A : A \subseteq [c_1, .., c_N]\}$ be a function. The function *access* satisfies the following three conditions:

(1) $\forall i \;:\; access(i) \neq \phi$.

(2) $\forall j \;:\; access(j) \not\subseteq \bigcup_{i<j} access(i)$.

(3) $\forall j \;:\; access(j) \not\subseteq \bigcup_{i>j} access(i)$.

if and only if the function *access* is a permutation (i.e. *access* is a one-to-one function in which for every $i : |access(i)| = 1$).

Proof: The necessity of these conditions is obvious. In this proof we show the sufficiency of these conditions. Let n be a function defined by: $n(j) \triangleq \left|\bigcup_{i \leq j} access(i)\right|$

First, we prove by induction that $n(j) \geq j$ for every j.

The induction basis, $j = 1$ follows immediately from condition (1). To prove the induction step, we show that $n(j+1) \geq n(j) + 1$. This is true because condition (2) implies that:

$$access(j) - \bigcup_{i<j} access(i) \quad \neq \quad \phi$$

Using the induction hypothesis, we conclude that $n(j) \geq j$.

On the other hand $n(N) \leq N$. Since the function n is strictly increasing, we conclude that $n(j) = j$, for every j.

Second, we prove by induction that $|access(j)| = 1$, for every j.

The induction basis, $j = 1$ follows immediately from the fact that $n(1) = 1$. Suppose $|access(i)| = 1$, for every $i < j$. We show that $access(j)$ is a singleton as well:

Suppose $|access(j)| > 1$. We deal with two cases:

case (1): $access(j) \;\bigcap\; \bigcup_{i<j} access(i) = \phi$.

case (2): $access(j) \;\bigcap\; \bigcup_{i<j} access(i) \neq \phi$.

If case (1) holds, then $n(j) > n(j-1)+1$, which implies that $n(j) > j$, a contradiction to the statement $n(j) = j$. If case (2) holds, then since $access(i)$ is a singleton for every $i < j$, there exists an address i which satisfies $access(i) \subseteq access(j)$. This contradicts condition (3).

We conclude that $|access(j)| = 1$, for every j.

The function *access* must be one-to-one, since if there exist two different addresses, i and j satisfying $access(i) = access(j)$, then conditions (2) and (3) are not satisfied. This completes the proof that the function *access* is a permutation.
□