[Proceeding] Automatic March tests generation for multi-port SRAMs


Availability: This version is available at: [http://porto.polito.it/1499997/](http://porto.polito.it/1499997/) since: January 2007

Publisher: IEEE

Published version: DOI:10.1109/DELTA.2006.17

Terms of use: This article is made available under terms and conditions applicable to Open Access Policy Article ("Public - All rights reserved"), as described at [http://porto.polito.it/terms_and_conditions.html](http://porto.polito.it/terms_and_conditions.html)

Porto, the institutional repository of the Politecnico di Torino, is provided by the University Library and the IT-Services. The aim is to enable open access to all the world. Please share with us how this access benefits you. Your story matters.

(Article begins on next page)
Automatic March Tests Generation for Multi-Port 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

Testing of Multi-Port (MP) SRAMs requires special tests since the multiple and simultaneous access can sensitize faults that are different from the conventional single-port memory faults. In spite of their growing use, few works have been published on testing MP memories. In addition, most of the published work concentrated only on two ports memories (i.e., 2P memories). This paper presents a methodology to automatically generate march tests for MP memories. It is based on generations of single port memory march test firstly, then extending it to test a generic MP SRAMs. A set of experimental results shows the effectiveness of the proposed solution.

1. Introduction

Multi-Port memories (MP) peculiarity is their capability of performing more than one operation simultaneously. Semiconductor MPs are composed of a unique array of memory cell and a p-port to access it ($p \geq 2$). Each port has an independent set of address, control, and data buses, making possible writing a value on a cell while another cell is being read. Multi-port SRAMs are nowadays widely used as embedded memories in a plenty of digital systems, like telecommunications ASICs and multiprocessor systems [1].

The problem of testing multi-port memories has been faced using and ad-hoc technique, without targeting specific functional fault models. In [2] [3] the authors assume that Single Port (SP) test algorithms provide a high fault coverage when applied to MP memories. The test methodology performs SP test algorithms on each port, but the deep fall of the effectiveness of the applied tests shows that ad-hoc fault models for MP must be adopted.

In [4] a new theoretical fault model (complex coupling fault) and its test solution are presented. Unfortunately the fault model is not validated by experimental analysis (i.e., it isn’t a realistic fault model), and the test complexity (i.e., the length of the test algorithm) is exponential w.r.t. the number of port: $O(n^p)$.

In [5] the authors present realistic fault models validated by industrial analysis. Taking into account the simultaneous access in memories, march tests were developed.

All the published tests solutions have been manually generated, a task that always requires a lot of time, expertise, and that sometimes does not succeeds in covering particularly complex memory faults.

Although several methodologies to automatize the march tests generations have been proposed [6] [7] [8] [9] [10], none of them faces the problem of the MP test.

In this paper, we present a systematic approach to automatically generate March Tests for MP SRAMs based on the tests generator engine presented in [10]. Moreover taxonomy of realistic fault models for generic p-port memories will be presented.

The paper is structured as follows: section 2 presents the proposed test generation methodology. In Section 3 a complete description of memory faults modeling will be exploit. Section 4 details the notation used to represent march tests for both single and multi port memories. In section 5 a detailed analysis of the methodology is presented, while section 6 provides experimental results that proof the efficiency of our approach. Section 7 summarizes the main contributions and future developments of this research.

2. The Proposed Test Generation Methodology

The adopted methodology relies on a formal model representing the fault behaviour (see Section 3).
To automatize the test generation phase we first generate the single port march test by resorting the march test generation tool published in [10].

The main steps of the methodology are:

(i) automatically translate the FPs to an “operational” representation of the faulty behaviour, referred to as Addresses FP, or AFP.

(ii) The original memory graph model is then automatically modified according to the AFP, to build the fault graph that is then traversed to generate the test. An efficient implementation has been done, profitably exploiting pruning conditions imposed by the goal of primarily generating March Test.

(iii) After the generation of Single Port (SP) March test, we apply the Multi Port translation able to extend the SP March in to MP march.

Each generated march test has been validated by simulation performed by memory fault simulator tool [11].

The overall generation methodology is summarized in Figure 1.

![Figure 1: Automatic MP march tests generation flow](image)

### 3. 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 (FC) 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 FCs.

In testing SRAMs, the stimuli to be applied are memory operations. When dealing with MP SRAMs, each stimulus could be applied on a different port. MP faults can thus be ranked into two main classes:

- **Strong fault**: a memory fault that can be **fully** sensitized by an operation; e.g., a single-port write or read operations fail, two simultaneous read operations fail, etc.

- **Weak fault**: a fault which is **partially** sensitized by an operation; e.g., due to a defect that creates a small disturbance of the voltage of the true node of the cell. However, a fault can be **fully sensitized** (i.e., become strong) when two or more weak faults are sensitized simultaneously, since their faults effect can be additive. This may occur when a MP operation is applied.

Fault modeling requires a rigorous formalism; 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 use \( n \) 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

\[
C = \{0^{[i]}, 1^{[i]}, \&^{[i]} | 0 \leq i \leq n-1\}
\]

where apex identifies the address of the cell. If the address is omitted, it means that the state could 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

\[
X = \{r^{[i]}_{d}, w^{[i]}_{d} | 0 \leq i \leq n-1; d \in (0,1)\} \cup \{t\}
\]

where:

- \( w^{[i]}_{d} \): a write operation of the value \( d \) performed in the cell \( i \);

- \( r^{[i]}_{d} \): a read operation performed in the cell \( i \). The value \( d \) it 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;

- \( t \): a wait operation for a defined period of time.

This additional element is needed to deal with Data Retention Faults [6].

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

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

**Definition 3:** A Sequence of conditions/operations (S) is the minimum sequence of stimuli and conditions of length \( m \) needed to sensitize the fault. The \( j \)-th condition/operation is represented as \( c[x] \), where \( c \in C \), and \( x \in X \).
Definition 4: A Fault Primitive FP represents the difference between an expected (fault-free) and the observed (faulty) memory behavior, denoted by:

\[ < SA ; SV / F / R > \]  

(3)

Where \( SA \) and \( SV \) are the set of \( S \) respectively applied to \( a \)-cell and \( v \)-cell, needed to sensitize the given fault.

Since \( S \) could be applied via several ports in parallel, \( SA \) and \( SV \) are represented as:

\[(S_1)^0 : (S_2)^1 : \ldots : (S_p)^p-1 \]  

(4)

The ";" denotes the fact that the sequences of operations (from 0 to \( p-1 \)) are applied simultaneously via the \( p \) ports. The apex denotes the target port.

\( F = \{ (f)^p \mid f \in C \} \) is the faulty behavior, i.e., the value (state) stored in the victim cells after applying \( S \). \( R = \{ (r)^n \mid r \in C \} \) is the sequence of values read on the aggressor cell when applying \( S \).

As an example FP = \(< 0w_1 : r_1 ; 0 / 1 / - >\) means that the operations ‘w1’ and ‘r1’ performed on the a-cell, trough the two ports, when the initial state is 0 for both a and v cells, causes the victim to flip. No addresses are specified; therefore this fault can affect each couple of memory cell.

The terminology of weak and strong faults is used in representing the MP FFMs as follow:

- **FP denotes a strong fault** represented by its FP, while \( wFP \) denotes the weak fault FP. For example, RDF denotes a strong Read Destructive Fault, while \( wRDF \) denotes a weak Read Destructive Fault.
- \( wFPi\&wFPj\& wFPk \): denotes a pPF consisting of \( p \) weak faults; \&" denotes the fact that the \( p \) faults in parallel (i.e., simultaneously) form the \( p \)-port fault (pPF). For example the \( wRDF\&wRDF\&wRDF \) denote a 3PF based on three weak RDFs [12].

Several FPs classification rules can be adopted, based on the number of memory operations \( (m) \) needed to sensitize the FP (e.g., static when \( m = 1 \) or dynamic fault elsewhere); or based on the number of memory cells (#FC) involved by the FP (e.g., single-cell where #FC = 1 or \( n \)-cells fault, elsewhere) [12].

3.1. Multi Port Constraints

As discussed in the previous section, a MP FFM requires the use of the ports to perform the sensitizing operations in parallel. Physical constraints impose some limitations on the set of allowed concurrent memory operations:

- simultaneous **write** operations are not allowed;
- simultaneous **read** operations are allowed;
As an example consider the follow March Test:

\[
\{ \hat{\theta} (w_1) \cup (r_1,w_0) \cup (r_0) \} \quad (6)
\]

Starting from (5) we can extended it to apply the operation simultaneously.

**Definition 6**: A MP March Test is defined as:

\[
MTG_{mp} = (N_p, S_p, P_p) \quad (7)
\]

where:

- \( N_p = N \cup \{\text{OPs}\} \) is the set of the nonterminal symbols;
- \( \sum_p = \sum \cup \{ '-', '.', 'n' \} \) is the set of terminal symbols (i.e., the alphabet). Don’t care ‘-’ denotes that any operation is allowed on the selected port, and ‘n’ denotes that no operations are allowed on the selected port;
- \( S_p = S \) is the start symbol. \( S_p \in N_p; \)
- \( P_p \subseteq N_p \times (N_p \cup \sum_p)^* \) is the set of productions detailed as follows:

\[
\begin{align*}
1) MT & \rightarrow '\text{ME}' \ D \\
2) ME & \rightarrow AO('OP')\|A'O('OP')\|ME \|e \\
3) OP & \rightarrow 'w'_{p0} \| 'r'_{p0} \| 'w'_{p0} \| 'r'_{p0} \| 'w'_{p0} \| 'r'_{p0} \| OP \\
4) AO & \rightarrow 'p' \| 'o' \| 'p' \| 'o' \\
5) D & \rightarrow 'p' \| 'o'
\end{align*}
\]

The march test (6) could be extended to MP test purpose as:

\[
\{ \hat{\theta} (w_1 : n) \cup (r_1 : \cdot , w_0 : r_1) \cup (r_0 : \cdot) \} \quad (8)
\]

This march test has been translated for two port memories (i.e., only two operations at each time are applied in parallel).

**5. Multi Port Translation**

The translation of a single port march test to a generic \( p \) Port march test is feasible under the constraints presented in Section 2.1. This phase requires as input the single port march test previously generated, and the number \( p \) of port (Figure 1)

The input march test has to be formatted by the march test generator phase in order to evidence the nature of the memory operations (i.e., by labeling each operations of the march test), that can be clustered in three categories:

1) **Initializing operations**: their can be only write operations;
2) **Sensitizing operations**: their could be either write or read operations;
3) **Observing operations**: their can be only read operations;

Note that an operation could be, at the same time, sensitize and observe the fault (i.e., read fault [13]) or initialize and sensitize the fault (i.e., state fault [13]).

This labeling procedure is done by the SP march test generator, where the information about each operations (i.e., if an operation is a sensitizing or initializing or observing) directly from the fault model (Section 3).

This phase corresponds to a set of rewrite rules, since the single port march test can be consider as a string accepted by the grammar defined in (5) where each symbol is a memory operation. Each rewrite rules is represented by the regular expression formalism [15].

Table 1 shows the rewrite rules, as an example if an operation is tagged “Sensitizing”, then rule #1 will be adopted. In case of multiple labelling (i.e., the operation is labelled both “Sensitizing” and “Observing”); the operator precedence has been implemented by the order of rewrite rules.

<table>
<thead>
<tr>
<th>#</th>
<th>Operation</th>
<th>Rewrite Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sensitizing</td>
<td>( w_d \rightarrow w_q : r_q : \cdots : r_x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( r_d \rightarrow r_q : r_q : \cdots : r_q )</td>
</tr>
<tr>
<td>2</td>
<td>Initializing</td>
<td>( w_d \rightarrow w_q : n : \cdots : n )</td>
</tr>
<tr>
<td>3</td>
<td>Observing</td>
<td>( r_d \rightarrow r_d : \cdot : \cdots : \cdot )</td>
</tr>
</tbody>
</table>

The rule having the highest precedence (#1 table 1) is that related to sensitizing operations, since we must add \( p-1 \) different operations to apply in parallel to fully sensitize the fault.

The problem of what kind of added operations (write or read) is solved by constraints detailed in Section 2.1. Only simultaneous \( p \) read operations are supported or one write and \( p-1 \) read operations are supported. Therefore rule #1 inserts \( p-1 \) read operations. The expected value to read from the memory cell \( x \) depends from the previous memory state.
6. Experimental results

This section reports some experimental results obtained applying the proposed algorithm to automatically generate March Tests to cover different sets of faults. We first generate march tests able to cover 3 port FMs detailed in [5] and here summarized for sake of readability.

FFMs involving one cell are:
- \( w_{\text{DRDF}}\&w_{\text{DRDF}}\&w_{\text{DRDF}} \) : applying three simultaneous read operations to the v-cell causes the cell to flip, but returning the correct values. (Deceptive Read Destructive Fault, DRDF);
- \( w_{\text{RDF}}\&w_{\text{RDF}}\&w_{\text{RDF}} \) : applying three simultaneous read operations to the v-cell causes the cell to flip, returning the incorrect value. (Read Destructive Fault, RDF)

FFMs involving two cells are:
- \( w_{\text{CFds}}\&w_{\text{CFds}}\&w_{\text{CFds}} \) : applying three simultaneous operations to the a-cell causes the cell to flip. (Disturb Coupling Fault, CFds)
- \( w_{\text{CFds}}\&w_{\text{DRDF}}\&w_{\text{DRDF}} \) : applying three simultaneous read operations to the v-cell causes the cell to flip if the a-cell is in a specific state, but returning the correct values.
- \( w_{\text{CFds}}\&w_{\text{RDF}}\&w_{\text{RDF}} \) : applying three simultaneous read operations to the v-cell causes the cell to flip if the a-cell is in a specific state, returning the incorrect values.

Consider as an example the \( w_{\text{CFds}}\&w_{\text{CFds}}\&w_{\text{CFds}} \) that is described by 8 FPs in Figure 2.

\[
\begin{align*}
\{w_1: x_2: x_0: 1 \rarrow 1\}, & \{w_1: x_2: 1 \rarrow 0\}; \{w_1: x_2: 1 \rarrow 0\}; \\
\{0: x_2: 1 \rarrow 0\}, & \{0: x_2: x_0: 1 \rarrow 1\}; \\
\{0: x_2: x_1: 0 \rarrow 1\}, & \{0: x_2: x_1: 0 \rarrow 1\}; \\
\{1: x_2: x_0: 0 \rarrow 1\}, & \{1: x_2: x_0: 0 \rarrow 1\}; \\
\{0: x_2: 1 \rarrow 1\}, & \{0: x_2: 1 \rarrow 1\}; \\
\{0: x_1: 1 \rarrow 1\}, & \{0: x_1: 1 \rarrow 1\}; \\
\{1: x_1: 0 \rarrow 1\}, & \{1: x_1: 0 \rarrow 1\}.
\end{align*}
\]

Figure 2 : \( x \in \{0,1\} \), \( d = \text{don't care} \)

The FFM is fully sensitized by the applications of the three weak faults on the different memory port. We generate first the SP march test covering the first FPs and summarized in Figure 3

\[
\begin{align*}
\{w_1: 1 \rarrow 1\}, & \{1: x_2: 1 \rarrow 1\}; \\
\{0: x_2: 1 \rarrow 0\}, & \{0: x_2: 1 \rarrow 0\}; \\
\{0: x_1: 1 \rarrow 1\}, & \{0: x_1: 1 \rarrow 1\}; \\
\{1: x_1: 1 \rarrow 1\}, & \{1: x_1: 1 \rarrow 1\}.
\end{align*}
\]

Figure 3 : single port FPs, \( x \in \{0,1\} \)

The generated SP march test is

\[
\{1: (w_1) \cup (r_1, w_0) \cup (r_0, w_1) \cup (r_1, w_0) \cup (r_0, w_1) \cup (r_1)\} \quad (9)
\]

After MP translation (i.e., applying the rewrite rules Table 1) we obtain:

\[
\{1: (w_1) \cup (r_1, r_2, w_3, r_4) \cup (r_0, r_2, r_3, w_4, r_5) \cup (r_0, r_2, r_3, w_4, r_6) \cup (r_0, r_2, r_3, w_4, r_7) \cup (r_1)\} \quad (10)
\]

That is able to cover \( w_{\text{CFds}}\&w_{\text{CFds}}\&w_{\text{CFds}} \) [5].

Table 3 shows the resulting March Tests. For each march test we report its complexity (length of march test) and the equivalent march test found in literature, and the targeted fault list, the last column shows the cpu time (in second). The algorithm has been implemented in about 900 lines of C++ code, compiled with gcc compiler. All the experiments are performed on an ASUS, AMD 1500Mhz based Laptop with 512 MB of RAM. Table 2 reports the fault list covered by each march test. The first four generate march tests have been already published [5], the last three are unknown, and \#7 (whose complexity is 22n) has the same structure of march SS [16]. It is able to detect all the static faults (one and two-cells) extensions for multiple-port memories. All generated March Tests have been verified using an ad hoc memory fault simulator [11] able to validate their correctness w.r.t. the target FP list. The fault simulator is also used to check the non-redundancy of each generated March Test.

7. Conclusion

This paper presented a methodology to automatically generate March Tests for multiple-port memories. A general model has been used to represent known memory static faults, and to possibly add new user-defined faults. The generation process stems from the generation of SP march tests, then properly translated into MP march tests by applying a set of rewrite rules. Experimental results have been presented to prove the applicability and the efficiency of the proposed approach. On going activities are focused on the automatic generation of MP march tests targeting additional classes of memory fault, including Dynamic and Linked Faults.

Table 2 : fault list

<table>
<thead>
<tr>
<th>#</th>
<th>Fault List</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>( w_{\text{DRDF}}&amp;w_{\text{DRDF}}&amp;w_{\text{DRDF}} )</td>
</tr>
<tr>
<td></td>
<td>( w_{\text{RDF}}&amp;w_{\text{RDF}}&amp;w_{\text{RDF}} )</td>
</tr>
<tr>
<td>#2</td>
<td>( w_{\text{CFds}}&amp;w_{\text{CFds}}&amp;w_{\text{CFds}} )</td>
</tr>
<tr>
<td>#3</td>
<td>( w_{\text{CFds}}&amp;w_{\text{DRDF}}&amp;w_{\text{DRDF}} )</td>
</tr>
<tr>
<td></td>
<td>( w_{\text{CFds}}&amp;w_{\text{RDF}}&amp;w_{\text{RDF}} )</td>
</tr>
<tr>
<td>#4</td>
<td>All the 3 port FFM</td>
</tr>
<tr>
<td>#5</td>
<td>All the single cell Static Fault</td>
</tr>
<tr>
<td>#6</td>
<td>All the CFds</td>
</tr>
<tr>
<td>#7</td>
<td>All static FFM</td>
</tr>
</tbody>
</table>
8. References


---

**Table 3 : experimental results**

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>O (n)</th>
<th>Known March Test</th>
<th>CPU time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>{ (w0,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>6n</td>
<td>3PF1</td>
<td>0.030</td>
</tr>
<tr>
<td>#2</td>
<td>{ (w_1,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>10n</td>
<td>3PF2a</td>
<td>0.028</td>
</tr>
<tr>
<td>#3</td>
<td>{ (w_1,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>13n</td>
<td>3PF2v</td>
<td>0.210</td>
</tr>
<tr>
<td>#4</td>
<td>{ (w_1,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>14n</td>
<td>3PF</td>
<td>0.204</td>
</tr>
<tr>
<td>#5</td>
<td>{ (w_1,\ldots) } { (w_0,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>9n</td>
<td>-</td>
<td>0.093</td>
</tr>
<tr>
<td>#6</td>
<td>{ (w_1,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>14n</td>
<td>-</td>
<td>0.201</td>
</tr>
<tr>
<td>#7</td>
<td>{ (w_1,\ldots) } { (w_0,\ldots) } { (r_0,\ldots) } { (r_1,\ldots) } { (\ldots) }</td>
<td>22n</td>
<td>-</td>
<td>0.212</td>
</tr>
</tbody>
</table>