Towards a Uniform Notation for Memory Tests

Ad J. van de Goor Aad Offerman
Ivo Schanstra
Section Computer Architecture & Digital Technique
Department of Electrical Engeneering
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: vdgoor@duteca.et.tudelft.nl

Abstract

Historically many ways of expressing memory tests have been used; varying from special notations to the use of general purpose programming languages. A notation, originally introduced for march tests in 1990, has been adopted and extended by many researchers.

This paper extends that notation, in a systematic way, to a memory test language which allows march tests, pseudo march tests (such as GALPAT) and tests involving topological neighborhoods (to cover pattern sensitive faults) to be expressed in a unified way. The syntax and semantics facilitate the specification of memory tests in a compact way and can be processed using standard tools such as LEX and YACC.

Keywords: *memory tests, test languages, march tests, neighborhoods.*

1 Introduction

Progress in science often has been made after an appropriate notation, which allows for an accurate, compact unambiguous way of describing the domain-specific problems and solutions, has been invented. Testing semiconductor memories is an evolving area of research where new contributions are being made; new fault models are introduced and new, or improved, tests are proposed to detect the faults of the proposed fault models. Historically, a variety of notations has been used to represent memory tests; e.g., march tests, such as the MATS+ test were represented in a multi-line way whereby the addressing was specified by the way the operations of a given march element were positioned in successive lines [12][1] (see upper part of Figure 1). A later notation [5] uses specific symbols to indicate the addressing order such that a much more compact notation results (Figure 1, lower part).

ED&TC '96
0-89791-821/96 $5.00

W0 R0W1 R1W0
W0 R0W1 R1W0
W0 R0W1 R1W0
W0 R0W1 R1W0

$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$

Figure 1: notations for the MATS+ test

For the representation of pseudo-march tests, such as GALPAT and Walking 1/0 [6], as well as for tests for neighborhood pattern sensitive faults (NPSFs) [6] a conventional programming language, such as Pascal or C [9], has traditionally been used. These languages do not have constructs such that properties and characteristics, particular to memory tests, are easy to grasp from the test representation in program form.

The memory model, used implicitly in fault models such as 2-coupling faults (CFs), is one-dimensional; i.e., the memory cell array consists of a one-dimensional vector of cells. This can be deduced from the fact that for CFs, involving a coupling and a coupled cell, only two topological cases are considered: the coupling cell has a *lower* or a *higher* address than the coupled cell. Intuitively, this is not an accurate model of the way the memory cell array is implemented.

Other non-march tests such as GALROW, GALCOL [6] tests, tests for imbalance faults [11], and tests for sense amplifier saturation faults [6], etc., assume, in agreement with the physical organization of the memory cell array, a two-dimensional memory model, distinguishing rows and columns.

Another simplification of the memory model was the assumption that the memory has an external width of a single bit. [4] introduces the notion of *backgrounds* to generalize march tests to cover B-bit

($B \geq 1$) wide memories. [13][14] extends the notation introduced in [5] to include tests for B-bit memories.

Another implicit assumption was that only single ported memories were used; hence, no mechanism to express multiple operations, parallel in time, was provided for. To a limited extend, tests for FIFO memories (which inherently are multiported memories) have been described [8][15] by extending the notation introduced in [5].

The effectiveness of the GALROW and GALCOL tests, the test for imbalance and sense amplifier saturation faults, as well as tests for NPSFs, have shown that the one-dimensional memory model is not acceptable; the addressing mechanism should be able to specify addressing orders in two dimensions as well as topological neighborhoods (for tests for NPSFs and for the Checkerboard test [6]).

The remainder of this paper is organized as follows. Section 2 evaluates alternative notations for expressing memory tests, Section 3 gives a detailed motivation and description of the proposed language, Section 4 gives examples of some well-known tests, while Section 5 concludes this paper.

2 Alternative notations

The new notation for memory tests has to provide a unified framework for expressing march tests, pseudo march tests (such as GALPAT and GALROW) and tests which involve topological neighborhoods (such as tests for NPSFs and the Checkerboard test). The notation (i.e. test language) should have natural subsets to be used for the simple cases, while the syntax and semantics of the language have to be such that:

- tests can be expressed in an easy, natural way
- tests can be expressed in a compact way
- the language should have primitives for expressing the essential parts of a test (i.e. the addressing orders and the operations)
- the syntax should encourage the specification of complete and correct tests
- automatic processing of the test should be possible using standard tools such as LEX (a lexical analyzer) [10] and YACC (yet another compiler compiler) [2].

For the selection of a memory test language (MTL) one could use an existing programming language. The advantage is that the syntax and semantics are well defined while, certainly in case of the C programming language [9], almost any operation can be expressed in an efficient way. However, most of the requirements, as stated above, are not satisfied using such a language. The widespread use of the notation introduced in [5] indicates the need for an MTL and the preference of such an MTL over a conventional programming language. Many authors [13][8][15], thereafter have extended that language to allow for a unified representation of some additional features required by their specific test.

It is the intend of this paper to present a general framework for an MTL, based on [5] such that existing march tests, pseudo march tests and tests involving topological neighborhoods can be expressed in a uniform, natural way.

3 The Memory Test Language (MTL)

The syntax of MTL is defined using a variant of the BNF notation (Backus-Naur Form). Each syntactical category is denoted by its 'name' enclosed in brackets '⟨...⟩' and defined as follows:

⟨name⟩ ::= ⟨expression1⟩ | ⟨expression2⟩ | ...

The '::=' symbol, which means: 'is defined as', is followed by an expression of the categories from which the syntactical category is composed. A '|' symbol denotes a choice: one of the elements has to be selected. Categories between curly braces '{...}' can be repeated zero or more times. In cases where there could be confusion between symbols used in the BNF notation, called the meta language, and the test language MTL, the test language symbols are underlined. The category '⟨*empty*⟩' is special: it is empty. It can be used in a syntactical category that, or from which parts, can be omitted. To improve readability, subscripts are used for addressing specifiers and indices in the examples (providing a notation, as well as a language).

The proposed MTL is described in the following sections. Section 3.1 gives the BNF notation of the MTL for standard march tests, followed by some examples. The following section extends that MTL for addressing in a two-dimensional memory model. In Section 3.3 addressing identifiers are introduced. The next two sections describe notations for multiport operations and operations on multi-bit words. Section 3.6 introduces the concept of tiles, used for pattern sensitive faults. Section 3.7, gives the syntax of global operations, which affect the whole memory cell array.

3.1 BNF for traditional march tests

A 'march test' is delimited by curly braces '{...}', and consists of a sequence of 'march elements', separated by semicolons ';'.

⟨march test⟩ ::= {⟨march element⟩ {;⟨march element⟩}}

Each 'march element' consists of a symbol denoting the addressing order ('⇑': up addressing order, assuming a linearly increasing address, '⇓': down addressing order, assuming a linearly decreasing address, '⇕': one of the two previous addressing orders), followed by, delimited by parentheses '(...), a sequence of 'operations', separated by comma's ','.

⟨march element⟩ ::= ⟨addressing order⟩ (⟨operation⟩{,⟨operation⟩})

⟨addressing order⟩ ::= ⇓|⇑|⇕

An 'operation' is an element of the following set: '$r0$' (read operation with expected value 0), '$r1$' (read operation with expected value 1), '$w0$' (write 0 operation), '$w1$' (write 1 operation).

⟨operation⟩ ::= $r0 \mid r1 \mid w0 \mid w1$

The operations in a march element will be applied consecutively to each cell before continuing to the next cell.

Two examples of commonly known march tests are:

- *MATS+*: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$
- *March B*: $\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$

3.2 Addressing in a two-dimensional memory model

The memory cell array consists of rows and columns, both having their specific properties and address decoders. Therefore, it is often not sufficient to use a one-dimensional test (e.g. a conventional march test). The MTL (Memory Test Language) proposed below is able to cope with a two-dimensional memory model, with the one-dimensional memory model being a natural subset.

⟨MTL test⟩ ::= {⟨march element⟩ {;⟨march element⟩}}

Three types of 'march elements' can be distinguished: the 'normal march element', the 'tile march element', which allows topological neighborhood patterns (tiles) to be written to the memory cell array (see Section 3.6), and 'global march elements', which specify operations having effect on the total memory cell array (see Section 3.7).

⟨march element⟩ ::= ⟨normal march element⟩ |
⟨tile march element⟩ |
⟨global march element⟩

A 'normal march element' consists of a one-dimensional march element or a two-dimensional march element. A one-dimensional march element, starting with a '1 dimensional addressing direction', and containing, delimited by parentheses '(...)', a sequence of 'sub march elements', separated by comma's ',', is used in the one-dimensional memory model. A two-dimensional march element, starting with a two-dimensional addressing direction ('row addressing direction' or 'column addressing direction'), is used in a two-dimensional memory model. A two-dimensional march element is different from an ordinary (one-dimensional) march element in that it specifies more explicitly the order in which the memory is addressed. It starts with a 'row addressing direction', denoted by one of the following symbols: '↓', '↑', '↕', or with a 'column addressing direction' denoted by one those: '→', '←', '↔'. This two-dimensional addressing direction is followed by, delimited by parentheses '(...)', a sequence of 'column sub march elements' (in the case of a 'row addressing direction') or 'row sub march elements' (in the case of a 'column addressing direction'), separated by comma's ','.

⟨normal march element⟩ ::=
⟨1 dimensional addressing direction⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨1 dimensional addressing direction⟩⟨addressing ID⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨row addressing direction⟩
(⟨column sub march element⟩
{,⟨column sub march element⟩}) |
⟨row addressing direction⟩⟨addressing ID⟩
(⟨column sub march element⟩
{,⟨column sub march element⟩}) |
⟨column addressing direction⟩
(⟨row sub march element⟩
{,⟨row sub march element⟩}) |
⟨column addressing direction⟩⟨addressing ID⟩
(⟨row sub march element⟩
{,⟨row sub march element⟩})

To each addressing sequence an 'addressing ID' can be assigned for later use (see Section 3.3).

Each 'row sub march element' or 'column sub march element' consists of a two-dimensional addressing direction complementary to the one used for the

two-dimensional march element in which it is defined, followed by, delimited by parentheses '(...)', a sequence of operations, separated by comma's ','. So, the 'row addressing directions' '$\downarrow$', '$\uparrow$', and '$\updownarrow$' are complementary to the 'column addressing directions' '$\rightarrow$', '$\leftarrow$', and '$\leftrightarrow$' (each two-dimensional addressing direction in the first set is complementary to each one in the second set). In this way each complete two-dimensional march element is assured to sweep the entire memory cell array; thereby satisfying, in part, the completeness requirement of a test.

⟨column sub march element⟩ ::=
⟨column addressing direction⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨column addressing direction⟩⟨addressing ID⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨time operation⟩

A 'time operation' can be specified to allow for tests for time-dependent faults, such as data retention faults [4] (see Section 3.7).

⟨row sub march element⟩ ::=
⟨row addressing direction⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨row addressing direction⟩⟨addressing ID⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨time operation⟩

⟨addressing direction⟩ ::=
⟨1 dimensional addressing direction⟩ |
⟨row addressing direction⟩ |
⟨column addressing direction⟩

⟨1 dimensional addressing direction⟩ ::= $\Updownarrow|\Downarrow|\Uparrow$

⟨row addressing direction⟩ ::= $\updownarrow|\downarrow|\uparrow$

⟨column addressing direction⟩ ::= $\leftrightarrow|\rightarrow|\leftarrow$

It is possible to specify more addressing sweeps of the memory in the second dimension. For example, the following two-dimensional march element addresses, from left to right, all columns. Every column is addressed three times: twice from bottom to top, and then once from top to bottom.

$\rightarrow(\uparrow(\ldots);\uparrow(\ldots);\downarrow(\ldots))$

'Normal march elements' (both one and two-dimensional) contain 'sub march elements' which can be nested; in this way operations in a 'normal march element' are not limited to only operate on the current word in the memory cell array. They may have their own 'addressing directions' (in the 'addressing specifier'), in order to allow for nested addressing sweeps of the complete memory or only the current row or column, as e.g. required for the read part of Walking 1/0 and GALPAT tests. The property of this 'addressing specifier' is that it may contain an expression of 'addressing IDs' which can be used to label the corresponding addressing sweep and/or to restrict the range of generated addresses (see Section 3.3). The syntax of an 'operation' is specified in Section 3.4.

⟨sub march element⟩ ::=
⟨addressing specifier⟩
(⟨sub march element⟩{,⟨sub march element⟩}) |
⟨operation⟩ | ⟨time operation⟩

3.3 Addressing identifiers

Addressing identifiers can be used to name (identify) a given addressing sequence for later use in the test and/or to restrict the range of addresses of an addressing sequence. The following constructs have been introduced for this:

1. Label an addressing sequence (i.e. the addressing sequence generated by a particular addressing direction).
   This is done by subscripting it with an identifier; e.g. $\Uparrow_a$. The subscript variable a, when used later on, takes on the value of the current address of the $\Uparrow_a$ 'addressing specifier'. A complete example of a labeled addressing sequence and the use of the identifier, later on, is given at the next item.

2. Skip a given base word; as e.g. required for Walking 1/0.
   The addressing sequence which has to skip the base word will be subscripted with '- identifier'. E.g., $\Uparrow_a$ $(\updownarrow_{\neg a}$ $(\ldots))$ means that for each base word a the row addressing sequence will address all words in the column of a, except the base word a itself.

3. Labeling a partial addressing sequence.
   E.g., $\Updownarrow_a$ $(\leftrightarrow_{b=\neg a}$ $(\ldots))$ means that the column addressing sequence is restricted to all columns except the column of word a. These column addresses are assigned to identifier b for later use.

⟨addressing specifier⟩ ::=
⟨addressing direction⟩ |
⟨addressing direction⟩⟨addressing ID⟩ |
⟨addressing direction⟩⟨addressing ID⟩
$= \neg$⟨addressing ID⟩ |
⟨addressing direction⟩ $-$ ⟨addressing ID⟩

⟨addressing ID⟩ ::= $a \mid b \mid \ldots z$

The addressing identifiers used to restrict an addressing sequence have to be defined in an addressing sequence with the same or a lower dimension, because the definition of an addressing identifier in a two-dimensional addressing sequence *cannot* be used to restrict the range of a one-dimensional addressing sequence. For example the following is illegal: $\Updownarrow (\leftrightarrow_b (\Updownarrow_{\neg b} (\ldots)))$.

For example, a test that fills the entire memory cell array with 0s, writes a 1 to a base cell, and then checks if in the column of the base cell every non base cell still has its correct value, would look like this: $\{\Updownarrow (w0); \Updownarrow_a (w1, \updownarrow_{\neg a} (r0), w0)\}$.

Furthermore, identifiers can be used in operations by subscripting the 'r' (read) or 'w' (write) symbols. The subscript consists of a single identifier (then this identifier should be assigned to a two-dimensional address), or an identifier pair: i.e. two identifiers separated by a comma ',', denoting the row and the column of a two-dimensional address.

The following test fills the memory with 0s. Then, a base cell walks through the memory cell array and is set to 1. For every base cell set to 1, every other cell in the memory cell array is read. Then the base cell is read before continuing to the next base cell.

$\{\Updownarrow (w0); \Updownarrow_a (w1, \Updownarrow_{\neg a} (r0), r1, w0)\}$

Identifiers used in a read or a write operation can also be used with an offset in order to allow neighborhoods of a base cell to be accessed.

In the following example all cells are set to 0. Then, a base cell set to 1 walks through the memory cell array. For each base cell a type 1 neighborhood (consisting of four cells) is read before continuing to the next base cell. The complete syntax of the indices is given in Section 3.5.

$\{\Updownarrow (w0); \Updownarrow_a (w1, r_{a,a+1}0, r_{a+1,a}0, r_{a,a-1}0, r_{a-1,a}0, w0)\}$

3.4 Multi-port operations

To test multi-ported memories, 'operations' can consist of several 'port operations', separated by colons ':'. All port operations in an operation will be applied to the memory simultaneously, through the associated ports.

⟨operation⟩ ::= ⟨port operation⟩{:⟨port operation⟩}

For example, a march element that writes a 0 through port 0 to each memory cell, and at the same time, checks through port 1 if the previous cell indeed has become 0, would be specified like this:

$\Uparrow_a (w0 : r_{a-1}0)$

3.5 Operations on B-bit words

Until now memories were addressing just one bit per address. However, most memories address a word that consists of B bits; where $B > 1$. To allow for operations on multi-bit word memories, a multi-bit word operation is defined. Such an operation will be an 'r' (read operation) or a 'w' (write operation) followed by one or more binary digits '0', '1' (thus the single-bit word operation already introduced in the march test notation is a special case of a multi-bit word operation). The first digit corresponds to the most significant bit (msb) and the last digit corresponds to the least significant bit (lsb).

⟨port operation⟩ ::= r⟨expected data⟩ | w⟨data⟩ |
r⟨offset⟩⟨expected data⟩ |
w⟨offset⟩⟨data⟩ | nop

An unused port in a port operation is denoted by a 'nop' operation (no operation).

⟨offset⟩ ::=
[⟨addressing ID & offset⟩] |
[⟨addressing ID & offset⟩,⟨addressing ID & offset⟩]

⟨addressing ID & offset⟩ ::=
⟨addressing ID⟩ |
⟨addressing ID⟩⟨operator⟩⟨integer⟩

In an offset only an identifier assigned to a row or a complete addressing (the row address will be derived) can be used in specifying the row, and only an identifier assigned to a column or a complete addressing (the column address will be derived) can be used in specifying the column.

⟨operator⟩ ::= + | −

⟨integer⟩ ::= ⟨digit⟩{⟨digit⟩}

⟨digit ::= 0 | 1 | ... 9

⟨expected data⟩ ::= ⟨binary word⟩

⟨data⟩ ::= ⟨binary word⟩

⟨binary word⟩ ::= ⟨binary digit⟩{⟨binary digit⟩}

⟨binary digit⟩ ::= 0 | 1

A B-bit word must be specified by 1 or B bits. A one-digit operation applied to a multi-bit word will be expanded to B identical bits (a 0 will be expanded to B 0s; a 1 will be expanded to B 1s). Other possible expansion methods, for example Marching and Walking data backgrounds for detection of intra-word coupling faults [13][14], should be specified explicitly.

3.6 Tile march elements

Using multi-bit word operations allows for using background patterns within a word. It is also possible to use patterns that cover more than just one word: these are called *tiles*. Tiles are required for tests for pattern sensitive faults. The memory cell array is assumed to be covered completely with tiles with a rectangular shape. Within each tile local operations are performed; these are read and/or write operations applied to a particular location within the tile.

For example, a traditional test that can only be specified using tiles is the Checkerboard test [6]. It divides the memory cell array into two groups of cells as shown in Figure 2. A 1 is written to all 1-cells

1	2	1	2
2	1	2	1
1	2	1	2
2	1	2	1

Figure 2: *checkerboard pattern.*

and a 0 to all 2-cells. After completion all cells are read. The whole process is then repeated with 0s in all 1-cells and 1s in all 2-cells. A tile with a height of 2 words and a width of 2 words is the minimum size required to contain the repetitive pattern needed for the Checkerboard test. The tile consists of four 1-bit words as shown in Figure 3. The complete specification of the Checkerboard test is given in Section 4.

2x2 tile

word 0,0	word 0,1
word 1,0	word 1,1

Figure 3: 2×2 *tile for Checkerboard test.*

A 'tile march element' consists of a one or two-dimensional addressing specification followed by the 'tile size', and the 'tile operation' between parentheses '(...)'. The addressing specification defines the sequence in which the tiles in the memory cell array are addressed. The 'tile size' specifies the tile's size in words.

⟨tile march element⟩ ::=
⟨1 dimensional addressing direction⟩⟨tile size⟩
(⟨tile operation⟩) | ⟨row addressing direction⟩
⟨column addressing direction⟩⟨tile size⟩
(⟨tile operation⟩) | ⟨column addressing direction⟩
⟨row addressing direction⟩⟨tile size⟩
(⟨tile operation⟩)

The 'tile size' specifies the height ('number of rows') and the width ('number of columns') of the tile between square brackets '[...]'.

⟨tile size⟩ ::=
[⟨number of rows⟩,⟨number of columns⟩]
⟨number of rows⟩ ::= ⟨integer⟩
⟨number of columns⟩ ::= ⟨integer⟩

A 'tile operation' consists of 'local operations', separated by comma's ','.

⟨tile operation⟩ ::=
⟨local operation⟩{,⟨local operation⟩}

A 'local operation' starts with an 'r' (read) or a 'w' (write) followed by the word address in the tile between square brackets '[...]' and finally the data word itself.

⟨local operation⟩ ::= r⟨location⟩⟨expected data⟩ |
w⟨location⟩⟨data⟩
⟨location⟩ ::= [⟨row offset⟩,⟨column offset⟩]
⟨row offset⟩ ::= ⟨integer⟩
⟨column offset⟩ ::= ⟨integer⟩

For example, when it is desired that a 0 is written into location $[0,0]$ and a 1 read from location $[1,2]$, the following specification will do:
$[2,3](w[0,0]0, r[1,2]1)$

Local operations implicitly assume the use of the first suitable port; combinations of tiles and multi-ported operations are not possible, since tiles are used to detect faults in the memory cell array and multi-ported operations address problems in the decoder logic.

To detect Neighbourhood Pattern Sensitive Faults (NPSFs), tests especially designed for this divide the memory into adjacent tiling groups. Figure 4 and Figure 5 show the tiling groups for type 1 and type 2 neighborhoods respectively. In traditional tests for

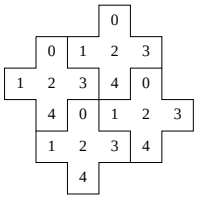

Figure 4: *type 1 tiling groups.*

NPSFs operations are applied to all cells having the same number.

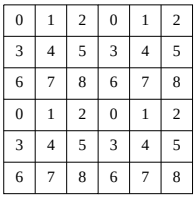

Figure 5: *type 2 tiling groups.*

For example, writing a 1 to all cells with number 5 in all type 2 tiling groups would be done with a tile operation like this: $\Updownarrow [3,3](w[1,2]1)$

To apply operations on a memory cell array divided into type 1 tiling groups, a rectangular tile should be determined in such a way that it is periodic both horizontally and vertically. In this case a 5×5 tile will do (see Figure 6).

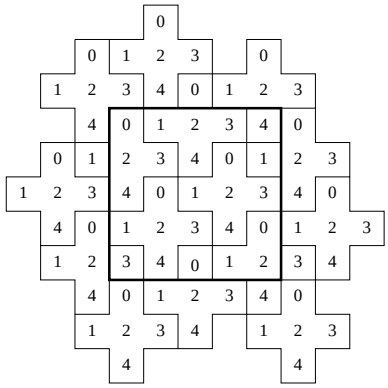

Figure 6: *tile for type 1 tiling groups.*

For example, a tile operation that reads a 0 from all words with the number 0 in the type-1 tiling groups, would look like this:

$\Updownarrow [5,5](r[0,0]0, r[1,3]0, r[2,1]0, r[3,4]0, r[4,2]0)$

In this way all necessary Neighborhood Patterns (NPs) can be applied to the memory cell array.

Reading and writing sequences (pseudo march tests) can be done too using tiles. For example, writing a pattern 011, notation: $w0 \mid 1 \mid 1$ to a memory cell array with two bits per word can be done by specifying a tile with a height of 1 word and a width of 3 words. The tile operation would look like this: $\Updownarrow [1,3](w[0,0]01, w[0,1]10, w[0,2]11)$

3.7 Global march elements

Global march elements contain operations that affect the entire memory. A 'global march element' can be a 'reset operation' or a 'time operation'.

⟨global march element⟩ ::= ⟨reset operation⟩ | ⟨time operation⟩

A 'reset operation', specified by an 'R', resets the memory: all parameters are set to their power-up value.

⟨reset operation⟩ ::= R

Some faults in the memory may take time to develop (e.g. data retention faults [4]). This requires time to elapse without read or write operations being applied to the memory. Until now each operation takes exactly one cycle. A 't' operation can be used to cause one cycle to pass without any operation being applied. A 't' operation followed by an integer can be used to specify the elapse of the number of cycles specified by that integer. A 'T' operation will cause enough time to pass as needed for all effects to become extinct.

⟨time operation⟩ ::= $t \mid t$⟨integer⟩ $\mid T$

For example, a march element that waits three memory cycles before reading 01 from each word of a two-bit word memory would be specified like this:

$\Updownarrow (t3, r01)$

4 Examples of some well-known tests

The constructs and expressive power of MTL will be demonstrated using well known memory tests.

- *MATS+* [12][1]:
  $\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$
- *Walking 1/0* [3]:
  $\{\Updownarrow (w0); \Updownarrow_a (w1, \Updownarrow_{-a} (r0), r1, w0); \Updownarrow (w1);$
  $\Updownarrow_a (w0, \Updownarrow_{-a} (r1), r0, w1)\}$

  The memory is filled with 0s. The base cell walks through the memory cell array and is set to 1. For every base cell set to 1, every other cell in the memory cell array is read. Then the base cell is read before continuing to the next base cell. After addressing the complete memory cell array the process is repeated with 1s in the memory cell array and a 0 in the base cell.
- The *GALROW* test [3] does the same thing as Walking 1/0, except that it reads the base cell after each read operation from its row:
  $\{\Updownarrow (w0); \Updownarrow_a (w1, \leftrightarrow_{-a} (r0, r_a1), w0); \Updownarrow (w1);$
  $\Updownarrow_a (w0, \leftrightarrow_{-a} (r1, r_a0), w1)\}$

- *Butterfly* [6]:

  $\{\Updownarrow (w0); \Updownarrow_a (w1, r_{a,a+1}0, r_{a+1,a}0, r_{a,a-1}0, r_{a-1,a}0, w0);$
  $\Updownarrow (w1); \Updownarrow_a (w0, r_{a,a+1}1, r_{a+1,a}1, r_{a,a-1}1, r_{a-1,a}1, w1)\}$

  All cells are set to 0. A base cell set to 1 walks through the memory cell array. For each base cell a type 1 neighborhood (consisting of four cells) is read before continuing to the next base cell. Then the complete process is repeated with the memory cell array set to 0 and the base cell set to 1.

- A *multi-ported* test:

  $\{\Uparrow_a (w0 : \text{nop} : r_{a-1}0); \Uparrow_a (w1 : r_{a+1}0 : r_{a-1}1);$
  $\Uparrow_a (w0 : r_{a+1}1 : \text{nop}); \Downarrow_a (w0 : \text{nop} : r_{a+1}0);$
  $\Downarrow_a (w1 : r_{a-1}0 : r_{a+1}1); \Downarrow_a (w0 : r_{a-1}1 : \text{nop})\}$

  This test walks through the memory cell array writing throught port 0, and reading through ports 1 and 2 from the cell that will be written next and the cell that just has been written.

- The *Checkerboard* test (van de Goor, 1991) can be specified using a tile with height 2 and width 2, which is the smallest possible rectangle that results in the checkerboard pattern when repeated. The algorithm looks like this:

  $\{\Updownarrow [2,2](w[0,0]1, w[0,1]0, w[1,0]0, w[1,1]1);$
  $\Updownarrow [2,2](r[0,0]1, r[0,1]0, r[1,0]0, r[1,1]1);$
  $\Updownarrow [2,2](w[0,0]0, w[0,1]1, w[1,0]1, w[1,1]0);$
  $\Updownarrow [2,2](r[0,0]0, r[0,1]1, r[1,0]1, r[1,1]0)\}$

- *March G* [7]:

  $\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1);$
  $\Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0); T; \Updownarrow (r0, w1, r1); T;$
  $\Updownarrow (r1, w0, r0)\}$

5 Conclusions

The proposed MTL has been defined using the well established syntax notation BNF of programming languages. This facilitated the standard tool LEX to be used for lexical analysis of a test specified in MTL, and the tool YACC to be used for parsing the MTL statements.

Primitive operations have been introduced to allow for a high level of abstraction and to reduce the semantic gap between the semantic world of the memory test designer and the capabilities of the MTL language.

The MTL constructs are based on the notation used for march tests which has already been adopted (and extended) by many researchers. It allows for a consistant, compact notation for march tests, pseudo march test, and tests for neighborhood pattern sensitive faults; in addition, the memory model is allowed to be two-dimensional; multi-ported and to contain B-bit ($B > 1$) words.

References

[1] Abadir, M.S. and Reghbati, J.K. (1983). Functional Testing of Semiconductor Random Access Memories. *ACM Computing Surveys*, **15**(3), pp. 175-198.

[2] Aho, A.V. and Johnson, S.C. (1974). LR Parsing. *Computing Surveys*, **6**(2), pp. 99-124.

[3] Breuer, M.A. and Friedman, A.D. (1976). *Diagnosis and Reliable Design of Digital of Digital Systems.* Computer Science Press, Inc., Woodland Hills, CA, USA.

[4] Dekker, R. et al. (1988). Fault Modelling and Test Algorithm Development for Static Random Access Memories. In *Proc. IEEE Int. Test Conference*, pp. 343-351.

[5] Goor, A.J. van de and Verruijt, C.A. (1990). An Overview of Deterministic Functional RAM Chip Testing. *ACM Computing Surveys*, **22**(1), pp. 5-33.

[6] Goor, A.J. van de (1991). *Testing Semiconductor Memories, Theory and Practice* (536 pages). John Wiley & Sons, Chichester, UK.

[7] Goor, A.J. van de (1993). Using March Tests to Test SRAMs. In *IEEE Design & Test of Computers, March 1993*, pp. 8-14.

[8] Goor, A.J. van de et al. (1995). Functional Test for Shifting-type FIFOs. In *Proc. IEEE European Test Conference*, Paris, March 6-9.

[9] Kernighan, B.W. and Ritchie, D.M. (1978). *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

[10] Lesk, M.E. (1975). Lex - a Lexical Analyzer Generator. *Computer Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill, N.J.

[11] Mazumder, P. (1988). Parallel Testing of Parametric Faults in a Three-Dimensional Random-Access Memory. In *Proc. IEEE Int. Test Conference*, pp. 933-941.

[12] Nair, R. (1979). Comments on "An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories". *IEEE Trans. on Computers*, **C-28**(3), pp. 258-261.

[13] Treuer, R.P. and Agarwal, V.K. (1993). Fault Location Algorithms for Repairable Embedded RAMs. In *Proc. of the IEEE Int. Test Conference*, pp. 825-834.

[14] Treuer, R.P. and Agarwal, V.K. (1993). Built-In Self-Diagnosis for Rapairable Embedded RAMs. *IEEE Design & Test of Computers*, **10**(2), pp. 24-33.

[15] Zorian, Y. et al. (1994). An Effective BIST Scheme for Ring-Address Type FIFOs. In *Proc. IEEE Int. Test Conference*, pp. 378-387.