# An Open Notation for Memory Tests

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

## Abstract

Historically many ways of expressing memory tests have been used, varying from the use of general purpose programming languages to special notations. 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 and open way, to a memory test language which allows march tests, pseudo march tests, tests involving topological neighborhoods (to cover pattern sensitive faults), line mode tests, and pseudo random tests, to be expressed in a unified manner. The syntax and semantics facilitate the specification of memory tests in a compact and readable way. Most important, the open structure allows extensions to the notation when necessary.

The notation presented in this paper is a sequel to an earlier, much more confined notation proposed in [van de Goor, 1996]. The complete OMTL language specification can be found in [Offerman, 1997].

**Keywords:** memory tests, test languages, march tests, neighborhoods, line mode tests, pseudo random tests.

## 1 Introduction

Progress in science often has been made after an appropriate notation, which allows for an accurate, compact, unambigious 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 fresh 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 [Nair, 1979 and Abadir, 1983] (see upper part of figure 1). A later notation [van de Goor, 1990] uses specific symbols to indicate the addressing order such that a much more compact notation results (figure 1, lower part).

```
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 [van de Goor, 1991], as well as for tests for neighborhood pattern sensitive faults (NPSFs) [van de Goor, 1991], a conventional programming language, such as Pascal or C [Kernighan, 1978], 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. Intuitively, this is not an accurate model of the way the memory cell array is implemented.

The effectiveness of non march tests, such as GALROW and GALCOL tests [van de Goor, 1991], tests for imbalance faults [Mazumder, 1988], and tests for sense amplifier saturation faults [van de Goor, 1991], 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.

Another implicit assumption was that only single-port 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 multi-port memories) have been described [van de Goor, 1995 and Zorian, 1994] by extending the notation introduced in [van de Goor, 1990].

Another simplification of the memory model was the assumption that the memory has an external width of a single bit. [Dekker, 1988] introduces the notion of *backgrounds* to generalize march tests to cover $B$-bit ($B \geq 1$) wide memories. [Treuer 1993 and 1993a] extends the notation introduced in [van de Goor, 1990] to include tests for $B$-bit memories.

The remainder of this paper is organized as follows. Section 2 evaluates alternative notations for expressing memory tests; section 3 introduces the traditional march tests; section 4 gives a detailed motivation and description of the language; section 5 gives examples of some well-known tests, while section 6 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), tests which involve topological neighborhoods (such as tests for NPSFs and the Checkerboard test [van de Goor, 1991]), line mode tests, and pseudo random tests. 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,
- the language should be extendable easily and in a natural way.

For the selection of an open memory test language (OMTL) 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 [Kernighan, 1978], 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 [van de Goor, 1990] indicates the need for an OMTL and the preference of such an OMTL over a conventional programming language. Many authors [Treuer, 1993, van de Goor, 1995, and Zorian, 1994] thereafter have extended that language to allow for a unified representation of some additional features required by their specific tests.

It is the intent of this paper to present a general framework for an OMTL, based on [van de Goor, 1990], such that existing march tests, pseudo march tests, tests involving topological neighborhoods, line mode tests, and pseudo random tests, can be expressed in a uniform, natural way. Furthermore, this language should allow others to define extensions for their own specific purposes.

## 3 Traditional march tests

A '*march test*' consists of a sequence of '*march elements*'. Each '*march element*' consists of a symbol denoting the '*addressing order*' ('$\Uparrow$': up addressing order, assuming an increasing address, '$\Downarrow$': down addressing order, assuming a decreasing address, '$\Updownarrow$': one of the two previous addressing orders), followed by a sequence of '*operations*'.

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

An example of a commonly known march tests is:

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

## 4 The Open Memory Test Language

In this section the syntax of OMTL is explained. Section 4.1 introduces addressing in a two-dimensional memory model. The next two sections describe notations for multi-port operations and operations on multi-bit words. In section 4.4 three-dimensional memories and their addressing are given. Section 4.5 introduces the concept of tiles, used for pattern sensitive faults. In section 4.6 a notation for the specification of line mode tests is introduced. The next section does this for pseudo random tests. Section 4.8 gives the syntax of global operations, which affect the whole memory cell array. In section 4.9 the sequential and parallel operators, which allow test parts to be repeated or to be executed in parallel, are presented.

## 4.1 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. The addressing method presented below is able to cope with a two-dimensional memory model, with the one-dimensional memory model being a natural subset. Furthermore, when necessary, the memory dimensions can be extended with new ones, as is done in section 4.4 for three-dimensional memory models.

An '*OMTL test*' consists of a sequence of '*march elements*'. Various '*march elements*' are defined: the '*normal march elements*', the '*tile march elements*', which allow topological neighborhood patterns (tiles) to be written to the memory cell array (see section 4.5), the '*line mode march elements*', to specify line mode tests (see section 4.6), the '*pseudo random march elements*', to specify pseudo random tests (see section 4.7), and the '*global march elements*', which specify operations having effect on the total memory (see section 4.8).

### 4.1.1 '*Normal march elements*'

Each '*normal march element*' consists of an '*addressing*', specifying the order in which the memory is addressed, followed by a sequence of '*sub march elements*'. These '*sub march elements*' 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, as e.g. required for the read part of the Walking 1/0 and GALROW tests (see section 5).

For example, getting ahead of the rest of this story, a test part that sets the memory cell array to all 0's, then sets a base cell to 1, and checks the column of this base cell before continuing to the next base cell (a GALCOL like test [Breuer, 1976]), would be specified like this:

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

In the nested addressing the addressing variable $a$ is used to exclude the current base cell from the column being checked for 0's. Addressing variables are introduced in section 4.1.4. The '*memory operations*' are discussed in section 4.2.

An '*addressing*' consists of a calligraphic $\mathcal{A}$, surrounded by an '*addressing type*', an '*addressing sequence*', and an '*addressing range*'.

$$\langle addressing \rangle ::= \qquad (1)$$
$${}_{\langle addressing\ type \rangle}\mathcal{A}^{\langle addressing\ sequence \rangle}_{\langle addressing\ range \rangle}$$

The addressing type, sequence, and range specify the exact way the memory cell array is addressed.

### 4.1.2 '*Addressing types*'

The '*addressing type*' is specified by a keyword, denoting the type and the dimension of the addressing.

### 4.1.3 '*Addressing sequences*'

An '*addressing sequence*' consists of more '*addressing sequence specifiers*', providing more information on the exact sequence in which the memory is addressed.

I.e. '*linear*' results in a linear addressing sequence, e.g. '$0, 1, 2, \ldots$'.

And the '*pingpong*' sequence results in addresses as depicted in figure 2.

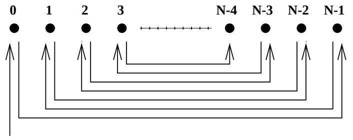

Figure 2: *pingpong addressing sequence.*

To improve the readability of test specifications and to meet the currently practiced notations, shortcuts for addressing notations are defined. The tables 1, 2, 3, and 4 give an overview.

Some examples are given in table 5.

| | |
|---|---|
| $\Updownarrow$ | ${}_{mca}\mathcal{A}^{\pm}$ |
| $\Uparrow$ | ${}_{mca}\mathcal{A}^{+}$ |
| $\Downarrow$ | ${}_{mca}\mathcal{A}^{-}$ |

Table 1: *Shortcuts for 'mca' addressing types.*

| | | | |
|---|---|---|---|
| $\updownarrow$ | ${}_{row}\mathcal{A}^{\pm}$ | $\leftrightarrow$ | ${}_{column}\mathcal{A}^{\pm}$ |
| $\uparrow$ | ${}_{row}\mathcal{A}^{+}$ | $\rightarrow$ | ${}_{column}\mathcal{A}^{+}$ |
| $\downarrow$ | ${}_{row}\mathcal{A}^{-}$ | $\leftarrow$ | ${}_{column}\mathcal{A}^{-}$ |

Table 2: *Shortcuts for 'row' and 'column' addressing types.*

#### 4.1.4 '*Addressing ranges*'

An '*addressing range*' consists of more '*addressing range specifiers*', allowing parts of the memory to be included or excluded from the addressing. To each addressing sequence an addressing variable can be assigned for later use.

Below we give an example of a base cell exclusion that is a variant of the one seen in section 4.1.1. First the memory cell array is filled with 0's. Then a base cell with value 1 walks through the memory. To keep track of its address, it is assigned to the addressing variable $a$. For each base cell, all cells in its row are checked for 0, except the base cell itself. After every check the base cell is read. The read operation subscripted with the addressing variable will be performed to the specified address, in this case address $a$.

$$\Updownarrow (w0); \Updownarrow_a (w1, \leftrightarrow_{\neg a} (r0, r_a1), w0)$$

More on the syntax of memory operations and subscripting these with operation addresses can be found in section 4.2 and 4.3.

An example using a range inclusion; writing a 16 × 16 block of 1's in a memory cell array of all 0's, as depicted in figure 3, is notated as:

$$\Updownarrow (w0); \Updownarrow_{32..47,16..31} (w1)$$

The pattern as in figure 3 can also be obtained by first writing all 1's to the memory cell array, and then writing 0's to it excluding the block:

$$\Updownarrow (w1); \Updownarrow_{\neg 32..47,16..31} (w0)$$

| | | | |
|---|---|---|---|
| $\nwsearrow$ | ${}_{diag0}\mathcal{A}^{\pm}$ | $\neswarrow$ | ${}_{diag1}\mathcal{A}^{\pm}$ |
| $\searrow$ | ${}_{diag0}\mathcal{A}^{+}$ | $\swarrow$ | ${}_{diag1}\mathcal{A}^{+}$ |
| $\nwarrow$ | ${}_{diag0}\mathcal{A}^{-}$ | $\nearrow$ | ${}_{diag1}\mathcal{A}^{-}$ |

Table 3: *Shortcuts for non-wrapped diagonal addressing types.*

| | | | |
|---|---|---|---|
| $\Nwsearrow$ | ${}_{diag0w}\mathcal{A}^{\pm}$ | $\Neswarrow$ | ${}_{diag1w}\mathcal{A}^{\pm}$ |
| $\Searrow$ | ${}_{diag0w}\mathcal{A}^{+}$ | $\Swarrow$ | ${}_{diag1w}\mathcal{A}^{+}$ |
| $\Nwarrow$ | ${}_{diag0w}\mathcal{A}^{-}$ | $\Nearrow$ | ${}_{diag1w}\mathcal{A}^{-}$ |

Table 4: *Shortcuts for wrapped diagonal addressing types.*

| | |
|---|---|
| $\Downarrow^{2linear}$ | ${}_{mca}\mathcal{A}^{-2linear}$ |
| $\uparrow^{2power4}$ | ${}_{row}\mathcal{A}^{+2power4}$ |
| $\leftrightarrow^{gray}$ | ${}_{column}\mathcal{A}^{\pm gray}$ |
| $\nwarrow^{pingpong}$ | ${}_{diag0}\mathcal{A}^{-pingpong}$ |
| $\Swarrow^{3linear}$ | ${}_{diag1w}\mathcal{A}^{+3linear}$ |

Table 5: *Example shortcut notations.*

## 4.2 Multi-port memory operations

To test multi-port memories, '*memory operations*' can consist of several '*port operations*', which will be applied to the memory simultaniously through the associated ports. If necessary, the '*port*' of a '*port operation*' can be specified explicitly by superscripting the '*operation*'.

For example, a march element that writes a 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:

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

## 4.3 Operations on multi-bit words

To allow for operations on multi-bit word memories, a multi-bit word '*operation*' is defined.

For example, a march element writing and checking a Marching 1 pattern [Treuer, 1993 and 1993a] in a 4-bit word memory can be specified like this:

$$\Uparrow^{linear} (w0000, r, w1000, r, w0100, r, w0010, r, w0001,$$
$$r, w0000)$$

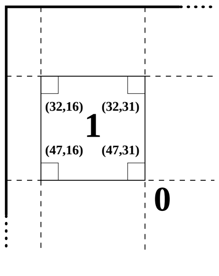

Figure 3: *a* 16 × 16 *block of* 1*'s in a memory cell array of all* 0*'s.*

The '*operation address*' subscript can be used to operate on other cells than the current one addressed, i.e. to access neighborhoods of a base cell. The possibility to do this is required by tests like Butterfly (see section 5).

## 4.4 Three-dimensional memories

Often the core of memories consists of more than one memory cell array. The correct array is selected by a third address decoder, the array decoder. Addressing the various arrays can be specified using an '*array addressing*'.

For example, a Zero-One or MSCAN test [Abadir, 1983] performed to all arrays sequentially is notated as:

$$\{_{array}\mathcal{A}^{+linear}(\Updownarrow (w0), \Updownarrow (r0), \Updownarrow (w1), \Updownarrow (r1))\}$$

## 4.5 Tiles

Tiles, used for patterns that cover more than just one word, are required for tests for Neighborhood Pattern Sensitive Faults (NPSF's). The memory cell array is assumed to be covered completely with tiles with a rectangular shape.

For example, a traditional test that can only be specified using tiles is the Checkerboard test [van de Goor, 1991]. It divides the memory cell array into two groups of cells as shown in figure 4. A 1 is written to all 1-cells 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 complete specification of the Checkerboard test is given in section 5.

| | | | |
|---|---|---|---|
| 1 | 2 | 1 | 2 |
| 2 | 1 | 2 | 1 |
| 1 | 2 | 1 | 2 |
| 2 | 1 | 2 | 1 |

Figure 4: *Checkerboard pattern.*

For example, a tile march element writing 0's to all $(0, 1)$ locations of a $2 \times 2$ tile pattern, first addressing the rows in positive direction and then addressing the column in negative direction, would be written as:

$$\uparrow\leftarrow^{[2,2]} (w_{[0,1]}0)$$

### 4.5.1 Special cases: type 1 and type 2 tiling groups

For the often used type 1 (consisting of 4 adjacent cells) and type 2 neighborhoods (consisting of 8 adjacent cells) two keywords '*type1*' and '*type2*' are defined that can be specified as the '*tiling groups type*' instead of the normal '*tile size*'. Now the '*location*' of a local operation consists of only one '*index*', specifying the cell number within the tiling group.

For example, a tile march element writing a 0 to all 4 and 5 locations in a type 2 tiling group pattern, first addressing all columns and then the row in positive direction, would be specified like this:

$$\leftrightarrow\uparrow^{[type2]} (w_{[4]}0, w_{[5]}0)$$

### 4.5.2 Defining new tiling groups

When necessary, new tiling groups with their own '*tiling group type*' keyword can be defined by specifying a '*tile size*' and a '*location numbering*'.

For example, to define the type 1 tiling groups, a $5 \times 5$ tile is required, resulting in a specification like this:

$$\begin{aligned} \mathit{type1} = {} & [5,5](0,1,2,3,4,\ 2,3,4,0,1,\ 4,0,1,2,3, \\ & 1,2,3,4,0,\ 3,4,0,1,2) \end{aligned}$$

And the checkerboard pattern as depicted in figure 4 would be specified as:

$$\mathit{checkerboard} = [2,2](1,2,\ 2,1)$$

## 4.6 Line mode tests

Some memory chip designs already have built-in facilities for testing purposes. These can be exploited using '*special operations*', or using "special" '*march elements*', as is the case, for example, with the '*pseudo random march elements*' (see section 4.7).
In this section '*line operations*', acting on complete rows in the memory cell array, are introduced [Inoue, 1987 and Matsuda, 1989].

A read '*line operation*' ('$_{line}r$') reads all bits in the current row, using an '*interleaving*' and '*offset*'. If the '*expected data*' is a 0, all the selected bits in the row are ORed; if it is a 1, all bits and ANDed. A write '*line operation*' ('$_{line}w$') writes all bits in the current row.

For example, the following march elements write and read 0's to and from the memory cell array, accessing complete lines, and using an interleaving of 4 and an offset of 2:

$$\updownarrow (_{line}w1); \updownarrow (_{line}w^{4,2}0, _{line}\, r^{4,2}0, _{line}\, w^{4,2}1)$$

## 4.7 Pseudo random tests

The '*addressing sequence*' of a pseudo random march element consists of two addressing sequence specifiers: the first specifying the '*pseudo random addressing sequence*', and the second specifying the '*pseudo random generator*'. A '*pseudo random addressing sequence*' can be an ordinary '*addressing sequence*', having the same syntax and meaning as the tile addressing sequences defined in section 4.5. Or it can be a '*pseudo random pseudo random addressing sequence*', in which the addresses are determined by taking a '*pseudo random pattern*' from the pseudo random generator. A '*pseudo random pattern*' is a string of 0's and 1's from which parts can be derived from the pseudo random string generated by the pseudo random generator.

For the next examples, suppose the pseudo random string currently has the value 11001010. Then the pseudo random pattern $?_7$ selects bit number 7 (note that the lesb has index 0), being a 1. $?_{4,1}$ selects the substring 0101.

An example combining with binary digits and overlapping selections: the pseudo random pattern $11?_4 00?_{6,4}$ results in 11000100.

The '*pseudo random generator*' can be specified by the '*seed*', the '*binary word*' that specifies the pseudo random string to start with, and a '*characteristic polynomial*'.

For example, a 4 bit pseudo random generator could look like this:

$$1000, x + x^3$$

Apart from a '$r$' (read) and '$w$' (write), a '*pseudo random operation*' can be a '*pseudo random operator*', in which case the operation itself is pseudo random. The '*pseudo random operator*' consists of a special symbol '$\frac{r}{w}$' subscripted with the '*index*' of the digit from the pseudo random string that will be driving the memory's R/W control line.

For example, a complete pseudo random march element with a pseudo random pseudo random addressing sequence for a 2 bit memory could look like this:

$$_{pseudo\ random}\mathcal{A}^{?_{4,0}?_{4,0};0111,x+x^2}(w?_{1,0}, r)$$

Another example, using the pseudo random operator:

$$_{pseudo\ random}\mathcal{A}^{+mca;1010,1+x^3}(w?_{2,1}, \frac{r}{w}_{2}?_{3,2}, r)$$

A memory with CRC logic (Cyclic Redundancy Check) and operating in compress mode works the outputs of pseudo random tests into a response signature. After completion the resulting response signature is compared against the value known to be correct. A '*pseudo random crc reset operation*', notated as '*crc R*', resets the pseudo random crc counter. A '*pseudo random crc check operation*' consists of the keyword '*crc*' followed by the '*expected data*'.

So, a typical CRC BIST (Build-In Self-Test) will have the following pattern:

$$crcR, \ldots, crc$$

## 4.8 Global operations

A '*global operation*' affect the entire memory.

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

Some faults in the memory may take time to develop, e.g. data retention faults [Dekker, 1988]. A '$d$' operation can be used to specify the elapse of one or more time cycles. Using the '$D$' operation, a certain amount of time can be passed. A '$D$' operation without any time specification will cause enough time to pass as needed for all effects to become extinct.

For example, a march element that waits three memory cycles before reading 11 from each word of a two-bit word memory would be specified like this:

$$\Updownarrow (3d, r11)$$

Setting the clock speed with which the memory is operated can be done using the '*clock operation*'.

For example, a test writing 1's to the memory array at a speed of $120MHz$, and then reading at $100MHz$, will be specified like this:

$$\{C120MHz; \Updownarrow (w1); C100MHz; \Updownarrow (r1)\}$$

Sometimes in a test, $V_{cc}$ is lowered to accelerate the appearing of certain faults. The '*Vcc operation*' can be used to specify $V_{cc}$.

For example, writing a 1 to a cell, lower $V_{cc}$ to $1.8V$ for $100ms$, and then check if the cell still contains the 1, would be specified like this:

$$\Updownarrow (w1, V_{cc}1.8V, D100ms, V_{cc}, r1)$$

Another method to stress a memory chip to issue certain faults is to increase the operating temperature. This can be specified using a '*temperature operation*'.

For example, a march element executed on a memory at $140°C$ could look like this:

$$T140C; \Updownarrow (w1, 100r1)$$

### 4.9 The sequential and parallel operators

Often in memory tests, similar looking parts can be recognized that only differ in addressing direction or have inversed data. To facilitate for shorter notations for these cases, a '*sequential operator*' is available. In its most general form, it consists of a calligraphic '$\mathcal{S}$' together with a running index and one or more expressions using this index. The resulting value is assigned to a '*variable*', a member of the lowercase Greek letters, and can be used in the test.

The sequential operator can also be used to define addressing sequences. For example, the pingpong sequence can be defined as follows:

$$pingpong = \mathcal{S}_{\alpha=0}^{\frac{1}{2}N-1} \alpha, N-1-\alpha$$

where $N$ is the length of the addressing dimension, resulting in the sequence $\mathcal{S}0, N-1, 1, N-2, 2, N-3, \ldots$.

Nowadays most memories contain dedicated hardware for testing purposes. Since the tests are often parallelized, furthermore a parallel operator is defined. It has the same syntax as the sequential operator, except the calligraphic '$\mathcal{P}$' replacing the '$S$'. The in this way specified test parts are executed in parallel.

For example, to run a test on all arrays of a three-dimensional memory simultaniously, the parallel operator can be used.

$$\mathcal{P}_{array}\{\ldots\}$$

## 5 Examples

In this section we will provide a set of examples to demonstrate the constructs and expressive power of OMTL. Where available, well known memory tests are used.

- *Sliding Diagonal* [van de Goor, 1991]:

  $$\{\Updownarrow (w0); \nesw\!arrow (\not\nwsearrow (w1), \Updownarrow (r), \not\nwsearrow (w0));$$
  $$\Updownarrow (w1) : \neswarrow (\not\nwsearrow (w0), \Updownarrow (r), \not\nwsearrow (w1))\}$$

  The memory cell array is filled with 0's. Then a diagonal is set to 1 and the complete memory cell array is checked. This is applied for each diagonal in the memory cell array. After completion the test is repeated with 0's in the diagonal and a background of all 1's.

- *Walking 1/0* [Breuer, 1976]:

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

  The memory is filled with 0's. 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 1's in the memory cell array and a 0 in the base cell.
  An alternative, shorter but probably less readable notation, using the sequential operator:

  $$\mathcal{S}_{\alpha=0,1}\{\Updownarrow (w\alpha); \Updownarrow_a (w\overline{\alpha}, \Updownarrow_{\neg a} (r\alpha), r\overline{\alpha}, w\alpha)\}$$

- *Butterfly* [van de Goor, 1991]:

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

  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.

- The *Checkerboard* test [van de Goor, 1991] can be specified using the checkerboard tile with height 2 and width 2. The algorithm looks like this:

  $$\{\Updownarrow^{[checkerboard]} (w_{[0]}1, w_{[1]}0);$$
  $$\Updownarrow^{[checkerboard]} (r_{[0]}1, r_{[1]}0);$$
  $$\Updownarrow^{[checkerboard]} (w_{[0]}0, w_{[1]}1);$$
  $$\Updownarrow^{[checkerboard]} (r_{[0]}0, r_{[1]}1)\}$$

## 6 Conclusions

The proposed OMTL is an extendable language and has been defined using the well established syntax notation BNF for programming languages. It allows for the use of a uniform notation for all memory tests. The language syntax is given in the BNF notation used for the unambigious definition of computer programming languages, and because of that lexical analysis and parsing methods from this world can be applied to memory tests.

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 OMTL language.

The OMTL 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 tests, tests for neighborhood pattern sensitive faults, line mode tests, and pseudo random tests. In addition, the memory model is allowed to be two- and three-dimensional, multi-port, and to contain $B$-bit ($B > 1$) words.

OMTL is an open notation. Memory test designers and researchers will be able to add extensions to fit their needs. In this document enough material has been provided to enable others to build on the notation proposed here in a orthogonal and consequent way.

The expressive power of OMTL has been demonstrated using many examples from a large variety of different classes of memory tests. Using OMTL, the communication between designers and implementors of memory tests will be more efficient and less error prone.

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

[3] 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.

[4] 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.

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

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

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

[8] Goor, A.J. van de, Offerman, A., and Schanstra, H.I. (1996). Towards a Uniform Notation for Memory Tests. In *Proc. European Design & Test Conference*, pp. 420-427.

[9] Inoue, J. et al. (1987). Parallel Testing Technology for VLSI Memories. In *Proc. IEEE Int. Test Conference*, pp. 1066-1071.

[10] Jonge, J.H. de and Smeulders, A.J. (1976). Moving Inversions Test Pattern is Thorough, Yet Speedy. *Computer Design*, May 1976, pp. 169-173.

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

[12] Matsuda, Y. et al. (1989). A New Array Architecture for Parallel Testing in VLSI Memories. In *Proc. IEEE Int. Test Conference*, pp. 322-326.

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

[14] 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.

[15] Offerman, A. and van de Goor, Ad. J. (1997). *An Open Notation for Memory Tests.* Technical Report No.1-68340-44(1997)07, Delft University of Technology.

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

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

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