An Open Notation for Memory Tests

Aad Offerman Ad J. van de Goor
Technical Report No.1-68340-44(1997)07
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 report is a sequel to an earlier, much more confined notation proposed in [van de Goor, 1996].

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 left 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, right 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. 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.
Non march tests, such as GALROW, GALCOL tests [van de Goor, 1991], tests for imbalance faults [Mazumder, 1988], and tests for sense amplifier saturation faults [van de Goor, 1991], etc., assume, in agreement with the physical organization of the memory cell array, a two-dimensional memory model, distinguishing rows and columns.
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 [van de Goor, 1991]).

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 BNF notation, used to specify the proposed language; 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), 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 The BNF notation

The syntax of OMTL is explained using a variant of the BNF notation (Backus-Naur Form). Each syntactical category is denoted by its '*name*' enclosed in brackets '$\langle \ldots \rangle$' and defined as follows:

$\langle name \rangle ::= \langle expression1 \rangle \mid \langle expression2 \rangle \mid \ldots$

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 OMTL, the test language symbols are underlined. The category '$\langle empty \rangle$' is special: it is empty. It can be used in a syntactical category that, or from which parts, can be omitted.

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 ';'.

$$\langle march\ test \rangle \quad ::= \quad \underline{\{}\langle march\ element \rangle \{\underline{;}\langle march\ element \rangle\}\underline{\}} \tag{1}$$

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, delimited by parentheses '(...)', a sequence of '*operations*', separated by comma's ','.

$$\langle march\ element \rangle \quad ::= \quad \langle addressing\ order \rangle \underline{(}\langle operation \rangle \{\underline{,}\langle operation \rangle\}\underline{)} \tag{2}$$

$$\langle addressing\ order \rangle \quad ::= \quad \Downarrow | \Uparrow | \Updownarrow \tag{3}$$

Note that the addressing orders are not necessarily linear, as long as the up addressing order '$\Uparrow$' and the down addressing order '$\Downarrow$' are each other's opposite.

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).

$$\langle operation \rangle \quad ::= \quad r0 \mid r1 \mid w0 \mid w1 \tag{4}$$

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

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.

Note that the BNF specification of OMTL is only an expedient used to specify the syntax of a memory test; tests complying with this specification are not necessarily semantically correct.

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 (e.g. a conventional march 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*' is delimited by curly braces '{...}', and consists of a sequence of '*march elements*', separated by semicolons ';'.

$$\langle \textit{OMTL test} \rangle \quad ::= \quad \underline{\{}\langle \textit{march element} \rangle \{\underline{;}\langle \textit{march element} \rangle\}\underline{\}} \tag{5}$$

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 and block 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).

$$\begin{aligned} \langle \textit{march element} \rangle \quad ::= \quad & \langle \textit{normal march element} \rangle \mid \\ & \langle \textit{tile march element} \rangle \mid \\ & \langle \textit{line mode march element} \rangle \mid \\ & \langle \textit{pseudo random march element} \rangle \mid \\ & \langle \textit{global march element} \rangle \mid \\ & \ldots \end{aligned} \tag{6}$$

When necessary, new '*march elements*' can be defined, or the existing notations can be extended.

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, delimited by parentheses '(...)', a sequence of '*sub march elements*', separated by comma's ','.

$$\langle \textit{normal march element} \rangle \quad ::= \quad \langle \textit{addressing} \rangle \underline{(}\langle \textit{sub march element} \rangle \{\underline{,}\langle \textit{sub march element} \rangle\}\underline{)} \tag{7}$$

'*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. They may have their own '*addressings*', in order to allow for nested addressing sweeps of the complete memory or parts of it, as e.g. required for the read part of the Walking 1/0 and GALROW tests (see section 5).

$$\begin{aligned} \langle \textit{sub march element} \rangle \quad ::= \quad & \langle \textit{addressing} \rangle \underline{(}\langle \textit{sub march element} \rangle \{\underline{,}\langle \textit{sub march element} \rangle\}\underline{)} \mid \\ & \langle \textit{memory operation} \rangle \mid \\ & \langle \textit{global operation} \rangle \end{aligned} \tag{8}$$

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.

Of course it is possible to specify more addressing sweeps of the memory in the second dimension. For example, the following march element addresses, in positive direction, all columns. Every column is addressed three times: twice in negative direction, and then once in positive direction.

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

The '*memory operations*' are discussed in section 4.2. Section 4.8 enlightens '*global operations*', which are part of the '*global march elements*' also. The rest of this section is devoted to the '*addressings*' of normal march elements.

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

$$\langle addressing \rangle \quad ::= \quad {}_{\langle addressing\ type \rangle}\mathcal{A}^{\langle addressing\ sequence \rangle}_{\langle addressing\ range \rangle} \tag{9}$$

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, placed at the lower left of the addressing symbol '$\mathcal{A}$'.

$$\begin{aligned} \langle addressing\ type \rangle \quad ::= \quad & mca \mid \\ & row \mid \\ & column \mid \\ & diag0 \mid \\ & diag1 \mid \\ & diag0w \mid \\ & diag1w \end{aligned} \tag{10}$$

The keyword denotes the type and the dimension of the addressing.

An '*mca*' addressing (memory cell array) is used with a one-dimensional memory model (see figure 2) and addresses the complete memory cell array. The rest of the addressing types assumes a two-dimensional

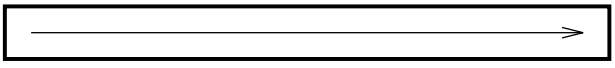

Figure 2: *1-dimensional memory model.*

memory model, as depicted in figure 3. The '*row*' and '*column*' addressing types address all rows and columns respectively. Addressing of the two diagonals can be specified using the '*diag0*' and '*diag1*' addressing types. '*diag0w*' and '*diag1w*' address the diagonals too, including the wrap-around part, as depicted in figure 4.

Unless explicitly specifying only a part of a memory test, the nesting of addressings in a two-dimensional test should result in a complete test, meaning that row as well as column addresses should be generated for each operation that needs these addresses.

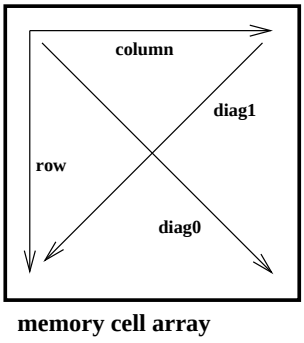

Figure 3: *2-dimensional memory model.*

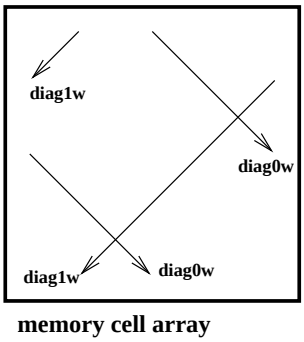

Figure 4: *Wrapped-around diagonals.*

4.1.3 '*Addressing sequences*'

In general, an '*addressing sequence*', placed at the upper right of the addressing symbol '$\mathcal{A}$', consists of more '*addressing sequence specifiers*', separated by semicolons ';'.

$$\langle \textit{addressing sequence} \rangle \quad ::= \quad \langle \textit{addressing sequence specifier} \rangle \{ ; \langle \textit{addressing sequence specifier} \rangle \} \tag{11}$$

The addressing sequence specifiers provide more information on the exact sequence in which the memory is addressed.

For normal march elements the '*addressing sequence specifier*' specifies the '*sequence*', the '*step*', and the '*direction*' of the generated addresses.

$$\langle \textit{addressing sequence specifier} \rangle_{\textit{normal march element}} \quad ::= \quad \langle \textit{direction} \rangle \langle \textit{step} \rangle \langle \textit{sequence} \rangle \tag{12}$$

The '*sequence*' is denoted by a keyword.

$$\begin{aligned} \langle \textit{sequence} \rangle \quad ::= \quad & \textit{linear} \mid \\ & \textit{2power} \langle \textit{factor} \rangle \mid \\ & \textit{gray} \mid \\ & \textit{pingpong} \mid \\ & \ldots \end{aligned} \tag{13}$$

'*linear*' results in a linear addressing sequence, e.g. '$0, 1, 2, \ldots$'.
The keyword '*2power*', possibly followed by a '*factor*', specifies addresses being a two power $2^{\langle factor \rangle \alpha}$.

$$\begin{aligned} \langle \textit{factor} \rangle \quad ::= \quad & \langle \textit{integer} \rangle \mid \\ & \langle \textit{empty} \rangle \end{aligned} \tag{14}$$

$$\langle integer \rangle \quad ::= \quad \langle digit \rangle \{ \langle digit \rangle \} \tag{15}$$

$$\langle digit \rangle \quad ::= \quad 0 \mid 1 \mid \ldots 9 \tag{16}$$

The factor defaults to 1, i.e. '$1, 2, 4, 8, \ldots$'. For example, '*2power*2' results in the addressing sequence '$1, 4, 16, \ldots$'. The '*gray*' keyword specifies a Gray code addressing sequence, in which each generated address differs only one bit from the previous address.
And the '*pingpong*' sequence results in addresses as depicted in figure 5.

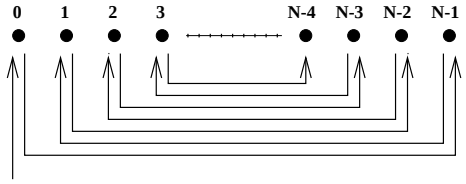

Figure 5: *pingpong addressing sequence.*

If required, the generated sequence can be stepped by specifying a '*step*' factor.

$$\begin{aligned} \langle step \rangle \quad ::= \quad & \langle integer \rangle \mid \\ & \langle empty \rangle \end{aligned} \tag{17}$$

For example, the addressing sequence '2*linear*' results in the addresses '$0, 2, 4, 6, \ldots$' being generated.

The '*direction*' can be increasing, denoted by a plus '+', decreasing, denoted by a minus '$-$', or unspecified, denoted by a plus-minus '$\pm$'.

$$\begin{aligned} \langle direction \rangle \quad ::= \quad & + \mid \\ & - \mid \\ & \pm \end{aligned} \tag{18}$$

Increasing and decreasing directions always generate opposite addressing sequences.

For example, the addressing sequences '$+gray$' and '$-gray$' result in the addresses '$0, 1, 3, 2, 6, 7, 5, 4$' and '$4, 5, 7, 6, 2, 3, 1, 0$' respectively (assuming an eight word length addressing dimension). The sequence '$\pm gray$' is one the above two sequences; which one is not specified.

Other addressing sequences can be defined when necessary. In most cases it is sufficient to define and describe new '*addressing sequence*' keywords to be used with the predefined addressing types of the existing march elements. Often this can be accompliced comfortably by using the sequence operator $\mathcal{S}$, discussed in section 4.9. Sometimes additional addressing sequence specifiers are required. Often these will have the form of an assignment. For example, an alternative way to specify a step in the generated addressing sequence could be to use a new addressing sequence specifier '*step*' like below:

$$\begin{aligned} \langle addressing\ sequence\ specifier \rangle_{step} \quad ::= \quad & step = \langle integer \rangle \mid \\ & \langle empty \rangle \end{aligned} \tag{19}$$

For example, using this feature the addressing '${}_{mca}\mathcal{A}^{\pm 2linear}$' could also be written as '${}_{mca}\mathcal{A}^{\pm linear;step=2}$'.
If it concerns a complete new category of tests, a new march element should be defined.

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*'

In general, an '*addressing range*', placed at the lower right of the addressing symbol '$\mathcal{A}$', consists of more '*addressing range specifiers*', separated by semicolons ';'.

$$\langle \textit{addressing range} \rangle \quad ::= \quad \langle \textit{addressing range specifier} \rangle \{ ; \langle \textit{addressing range specifier} \rangle \} \mid \langle \textit{empty} \rangle \tag{20}$$

The addressing range allows parts of the memory to be included or excluded from the addressing. If omitted, the addressing is complete.

For normal march elements to each addressing sequence an addressing variable can be assigned for later use. An '*addressing variable assignment*' consists of an '*addressing variable*' followed by an equal sign '='; then follow the '*addressing range specifiers*'.

$$\langle \textit{addressing range} \rangle_{\textit{normal march element}} ::= \langle \textit{addressing variable assignment} \rangle \langle \textit{addressing range specifier} \rangle \{ ; \langle \textit{addressing range specifier} \rangle \} \tag{21}$$

$$\langle \textit{addressing variable assignment} \rangle \quad ::= \quad \langle \textit{addressing variable} \rangle = \mid \langle \textit{empty} \rangle \tag{22}$$

$$\langle \textit{addressing variable} \rangle \quad ::= \quad a \mid b \mid \ldots z \tag{23}$$

If the '*addressing range*' contains only an '*addressing variable assignment*' and no '*addressing range specifiers*', the equals sign '=' can be omitted.

An '*addressing range specifier*' can be a '*base cell exclusion*', a '*range inclusion*', or a '*range exclusion*'.

$$\langle \textit{addressing range specifier} \rangle \quad ::= \quad \langle \textit{base cell exclusion} \rangle \mid \langle \textit{range inclusion} \rangle \mid \langle \textit{range exclusion} \rangle \mid \langle \textit{empty} \rangle \tag{24}$$

A '*base cell exclusion*' starts with a negation sign '¬' and consists of one '*expression*', specifying the complete '*address*' of the to be excluded base cell, or two '*expressions*', separated by a comma ',', specifying the '*row*'

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

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

$\searrow\!\!\!\searrow$	$diag0w\mathcal{A}^{\pm}$	$\swarrow\!\!\!\swarrow$	$diag1w\mathcal{A}^{\pm}$
$\searrow$	$diag0w\mathcal{A}^{+}$	$\swarrow$	$diag1w\mathcal{A}^{+}$
$\nwarrow\!\!\!\nwarrow$	$diag0w\mathcal{A}^{-}$	$\nearrow\!\!\!\nearrow$	$diag1w\mathcal{A}^{-}$

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

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

Table 5: *Example shortcut notations.*

and '*column*' of the to be excluded cell.

$$\langle base\ cell\ exclusion\rangle ::= \neg\langle address\rangle \mid \neg\langle row\rangle,\langle column\rangle \tag{25}$$

$$\langle address\rangle ::= \langle expression\rangle \tag{26}$$

$$\langle row\rangle ::= \langle expression\rangle \tag{27}$$

$$\langle column\rangle ::= \langle expression\rangle \tag{28}$$

The base cell exclusion of a one-dimensional addressing requires an expression specifying a complete address. To specify a row, only an expression specifying a row address or an expression specifying a complete address (the row address will be derived) can be used. In the same way, to specify a column, only an expression specifying a column address or an expression specifying a complete address (the column address will be derived) can be used.

For example, the following is illegal:

$$\Updownarrow (\leftrightarrow_a (\Updownarrow_{\neg a} (\ldots)))$$

Column address a does not define a base cell address in a one-dimensional addressing of the memory cell array.

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.

A '*range inclusion*' can be specified by an '*address range*', consisting of two '*expressions*' and a range sign '..' in between, denoting the borders of a one-dimensional range. Or by two ranges, specifying a '*row range*' and '*column range*' in a two-dimensional memory.

$$\langle range\ inclusion\rangle ::= \langle address\ range\rangle \mid \langle row\ range\rangle,\langle column\ range\rangle \tag{29}$$

$$\langle address\ range\rangle ::= \langle expression\rangle..\langle expression\rangle \tag{30}$$

⟨*row range*⟩ ::= ⟨*expression*⟩..⟨*expression*⟩ (31)

⟨*column range*⟩ ::= ⟨*expression*⟩..⟨*expression*⟩ (32)

For example, writing a 16×16 block of 1's in a memory cell array of all 0's, as depicted in figure 6, is notated as:

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

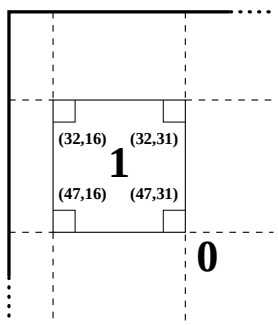

Figure 6: *a* 16×16 *block of* 1*'s in a memory cell array of all* 0*'s.*

A '*range exclusion*' is expressed by preceding a '*range inclusion*' with a negation sign '¬'.

⟨*range exclusion*⟩ ::= ¬⟨*range inclusion*⟩ (33)

Obvious but for completeness: depending on the first addressing range specifier, the complete memory is included or excluded at start. If the first addressing range specifier is a base cell exclusion or a range exclusion, the complete memory is included initially; if it is an inclusion, the complete memory is excluded at start.

So, for example, the pattern as in figure 6 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:

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

The syntax of an '*expression*' is not further specified. It should be an arithmetic expression, in which the addressing variables defined in the current scope can be used. Depending on the desired result (a complete address in a one-dimensional memory, or a row or column address in a two-dimensional memory), the complete address of the addressing variables or only the row or column part is used.

⟨*expression*⟩ ::= ... (34)

Recognize that it is possible to specify addresses that do not exist in the memory. For now, invalid operations are simply not applied. If other behaviour is required, for example a wrap-around of the addresses, a new global operation (see section 4.8) should be defined.

4.2 Multi-port memory operations

To test multi-port memories, '*memory operations*' can consist of several '*port operations*', separated by colons ':'. All port operations in a memory operation will be applied to the memory simultaniously through the associated ports, starting with port 0. If the same set of port operations should be repeated, these can be preceded by an '*integer*' specifying the number of '*times*'.

⟨*memory operation*⟩ ::= ⟨*times*⟩⟨*port operation*⟩{:⟨*port operation*⟩} (35)

⟨*times*⟩ ::= ⟨*integer*⟩ | (36)
⟨*empty*⟩

If necessary, the '*port*' of a '*port operation*' can be specified explicitly by superscripting the '*operation*'. If no port is specified, the operation is applied to the next port.

$$\langle port\ operation\rangle \quad ::= \quad \langle operation\rangle^{\langle port\rangle} \tag{37}$$

$$\begin{aligned}\langle port\rangle \quad ::= \quad & \langle integer\rangle \mid \\ & \langle empty\rangle\end{aligned} \tag{38}$$

For example, the *Towalk2* test walks a base cell with value 1 through a memory cell array containing all 0's. For each base cell, the base cell itself is checked, a 1 is written and then read back 16 times, before continuing with the next base cell. Then, the test is repeated using a base cell set to 0 in a memory cell array containing all 1's.

$$\{\Updownarrow (w0); \Updownarrow (r0, w1, 16r1, w0); \Updownarrow (w1); \Updownarrow (r1, w0, 16r0, w1)\}$$

Another 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:

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

And another example. This march element writes a 0 and then writes a 1 thirty times to the next word using port 3, while at the same time checking if the current word still contains a 0.

$$\Uparrow_a^{linear} (w0, 30r0 : w_{a+1}^3 1)$$

4.3 Operations on multi-bit words

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 '*w*' (write operation), followed by one or more '*binary digits*' '0', '1' specifying the '*expected data*' or the to be written '*data*'. So the single-bit word operation already introduced in the march test notation (see section 3.1) is a special case of the multi-bit word operation. The first digit corresponds to the most significant bit (mosb) and the last digit corresponds to the least significant bit (lesb). In case of a read operation the expected data may be omitted. An unused port in a memory operation is denoted by a '*nop*' operation.
The '*operation address*' subscript can be used to operate on other cells than the current one addressed. The possibility to do this is required by tests like Butterfly. An operation address can consist of one expression specifying an '*address*' in a one-dimensional memory, or two expresssions specifying the '*row*' and '*column*' address in a two-dimensional memory.

$$\begin{aligned}\langle operation\rangle \quad ::= \quad & r_{\langle operation\ address\rangle}\langle expected\ data\rangle \mid \\ & w_{\langle operation\ address\rangle}\langle data\rangle \mid \\ & nop\end{aligned} \tag{39}$$

$$\begin{aligned}\langle operation\ address\rangle \quad ::= \quad & \langle address\rangle \mid \\ & \langle row\rangle,\langle column\rangle \mid \\ & \langle empty\rangle\end{aligned} \tag{40}$$

$$\begin{aligned}\langle expected\ data\rangle \quad ::= \quad & \langle binary\ word\rangle \mid \\ & \langle empty\rangle\end{aligned} \tag{41}$$

$$\langle data\rangle \quad ::= \quad \langle binary\ word\rangle \tag{42}$$

$$\langle binary\ word\rangle \quad ::= \quad \langle binary\ digit\rangle\{\langle binary\ digit\rangle\} \tag{43}$$

$$\langle \textit{binary digit} \rangle \quad ::= \quad 0 \mid 1 \tag{44}$$

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 0's; a 1 will be expanded to B 1's). Other possible expansion methods, for example Marching and Walking data backgrounds for detection of intra-word coupling faults [Treuer, 1993 and 1993a], should be specified explicitly, or require an extension to the OMTL specification language.

For example; a march element writing and checking a Marching 1 pattern in a 4-bit word memory can be specified like this:

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

With an offset operation, addresses can be used to access neighborhoods of a base cell. 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 cell a type 1 neighborhood (consisting of four cells) is read before continuing to the next base cell.

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

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*', which addressing type is '*array*'. Further, it has the same syntax and possibilities as the normal addressings.

$$\langle \textit{array addressing} \rangle \quad ::= \quad {}_{array}\mathcal{A}^{\langle \textit{addressing sequence} \rangle}_{\langle \textit{addressing range} \rangle} \underline{(\ldots)} \tag{45}$$

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

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

Since the main purpose of a three-dimensional implementation is to speed up test time using the available parallellism, arrays will mostly not be addressed sequentially, but will be tested in parallel. Then the parallel operator discussed in section 4.9 can be used.

4.5 Tiles

Using multi-bit word operations allows for specifying 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 Neighborhood Pattern Sensitive Faults (NPSF's). 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 [van de Goor, 1991]. It divides the memory cell array into two groups of cells as shown in figure 7. 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 7: *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

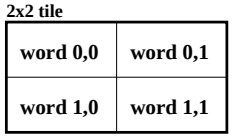

Figure 8: 2×2 *tile for the Checkerboard test.*

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 one-bit words as shown in figure 8. The complete specification of the Checkerboard test is given in section 5.

A '*tile march element*' consists of an addressing specification, followed by a '*tile operation*' between parentheses '(...)'. The tile addressing specification '$_{tile}\mathcal{A}$' is accompanied by an '*addressing sequence*' and an '*addressing range*'

$$\langle \textit{tile march element} \rangle \quad ::= \quad {}_{tile}\mathcal{A}^{\langle \textit{addressing sequence} \rangle}_{\langle \textit{addressing range} \rangle} \underline{(} \langle \textit{tile operation} \rangle \underline{)} \tag{46}$$

The '*addressing sequence*' of a '*tile march element*' consists of a '*tile addressing sequence*' followed by the '*tile size*'.

$$\langle \textit{addressing sequence} \rangle_{\textit{tile march element}} \quad ::= \quad \langle \textit{tile addressing sequence} \rangle \langle \textit{tile size} \rangle \tag{47}$$

The '*tile addressing sequence*' can be one-dimensional (specified by the keyword '*mca*' preceded by a '*direction*') or two-dimensional (specified by '*row*' and '*column*' keywords).

$$\begin{aligned} \langle \textit{tile addressing sequence} \rangle \quad ::= \quad & \langle \textit{1 dimensional tile addressing sequence} \rangle \mid \\ & \langle \textit{2 dimensional tile addressing sequence} \rangle \end{aligned} \tag{48}$$

$$\langle \textit{1 dimensional tile addressing sequence} \rangle \quad ::= \quad \langle \textit{direction} \rangle \textit{mca} \tag{49}$$

$$\begin{aligned} \langle \textit{2 dimensional tile addressing sequence} \rangle \quad ::= \quad & \langle \textit{direction} \rangle \textit{row} \underline{,} \langle \textit{direction} \rangle \textit{column} \mid \\ & \langle \textit{direction} \rangle \textit{column} \underline{,} \langle \textit{direction} \rangle \textit{row} \end{aligned} \tag{50}$$

Note the resemblances and differences with the addressing sequence of the normal march elements. The addressing of tile march elements is always linear.
The '*tile size*' specifies, between square brackets '[...]', the height ('*number of rows*') and width in words ('*number of columns*') of the tile.

$$\langle \textit{tile size} \rangle \quad ::= \quad \underline{[} \langle \textit{number of rows} \rangle \underline{,} \langle \textit{number of columns} \rangle \underline{]} \tag{51}$$

$$\langle \textit{number of rows} \rangle \quad ::= \quad \langle \textit{integer} \rangle \tag{52}$$

$$\langle \textit{number of columns} \rangle \quad ::= \quad \langle \textit{integer} \rangle \tag{53}$$

For example, a tile march element writing a 1 to all $(0,0)$ and $(1,2)$ locations of a 2×3 tile pattern, would be specified by a notation like this:

$${}_{tile}\mathcal{A}^{\pm mca[2,3]}(w_{[0,0]}1, w_{[1,2]}1)$$

Again we introduce shorter notations for the addressing specifications, which look very similar to the ones already introduced in section 4.1. A one-dimensional tile addressing can also be specified using a '$\Updownarrow$', '$\Uparrow$', '$\Downarrow$' symbol; a two-dimensional tile addressing can be specified using the '$\updownarrow$', '$\uparrow$', '$\downarrow$' and '$\leftrightarrow$', '$\rightarrow$', '$\leftarrow$' symbols together.

So the above example could also be written as:

$\Updownarrow^{[2,3]}\ (w_{[0,0]}1, w_{[1,2]}1)$

A tile march element writing 0's to all $(0,1)$ locations of a 2×2 tile pattern (as e.g. needed for the Checkerboard test), first addressing the rows in positive direction and then addressing the column in negative direction, would be written as:

$_{tile}\mathcal{A}^{+row,-column[2,2]}(w_{[0,1]}0)$

but also, using the shorter notation, as:

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

The '*addressing range*' of a '*tile march element*' is a restricted version of the addressing range of a normal march element. The base cell exclusions and one-dimensional range exclusions and inclusions are left out.

$$\langle \textit{addressing range} \rangle_{\textit{tile march element}} \quad ::= \quad \langle \textit{addressing range specifier} \rangle \{ \underline{;} \langle \textit{addressing range specifier} \rangle \} \tag{54}$$

$$\begin{aligned} \langle \textit{addressing range specifier} \rangle \quad ::= \quad & \langle \textit{tile range inclusion} \rangle \mid \\ & \langle \textit{tile range exclusion} \rangle \mid \\ & \langle \textit{empty} \rangle \end{aligned} \tag{55}$$

$$\langle \textit{tile range inclusion} \rangle \quad ::= \quad \langle \textit{row range} \rangle \underline{,} \langle \textit{column range} \rangle \tag{56}$$

$$\langle \textit{tile range exclusion} \rangle \quad ::= \quad \neg \langle \textit{tile range inclusion} \rangle \tag{57}$$

In above examples we have already seen some of the syntax of the tile operations. Here follows the formal specification. A '*tile operation*' consists of '*local operations*', separated by comma's ','.

$$\langle \textit{tile operation} \rangle \quad ::= \quad \langle \textit{local operation} \rangle \{ \underline{,} \langle \textit{local operation} \rangle \} \tag{58}$$

A '*local operation*' starts with an 'r' (read) or 'w' (write), subscribed with the address ('*location*') of the word within the tile, followed by the '*data*' word.

$$\begin{aligned} \langle \textit{local operation} \rangle \quad ::= \quad & r_{\langle \textit{location} \rangle} \langle \textit{expected data} \rangle \mid \\ & w_{\langle \textit{location} \rangle} \langle \textit{data} \rangle \end{aligned} \tag{59}$$

The '*location*' to which the local operation should be applied consists of the '*row offset*' and '*column offset*' within the tile.

$$\langle \textit{location} \rangle \quad ::= \quad \underline{[} \langle \textit{row offset} \rangle \underline{,} \langle \textit{column offset} \rangle \underline{]} \tag{60}$$

$$\langle \textit{row offset} \rangle \quad ::= \quad \langle \textit{integer} \rangle \tag{61}$$

$$\langle \textit{column offset} \rangle \quad ::= \quad \langle \textit{integer} \rangle \tag{62}$$

Local operations implicitly assume the use of the first suitable port; combinations of tiles and multi-port operations are neither possible nor required, since tiles are used to detect faults in the memory cell array and multi-port operations address problems in the decoder logic.

Reading and writing sequences (pseudo march tests) can be done too using tiles. For example, writing a pattern 011, notation: $w0 \mid 1 \mid 1$, can be done by specifying a tile with a height of 1 word and a width of 3 words. The tile march element would look like this:

$\Updownarrow^{[1,3]}\ (w_{[0,0]}0, w_{[0,1]}0, w[0,2]1)$

To do the same to a memory cell array with two bits per word, a 1×3 tile containing 3 words of 2 bits is required:

$\Updownarrow^{[1,3]}\ (w_{[0,0]}01, w_{[0,1]}10, w[0,2]11)$

4.5.1 Special cases: type 1 and type 2 tiling groups

To detect NPSF's, tests especially designed for this divide the memory into adjacent tiling groups. Figure 9 and 10 show the tiling groups for often used type 1 and type 2 neighborhoods respectively. In traditional tests for

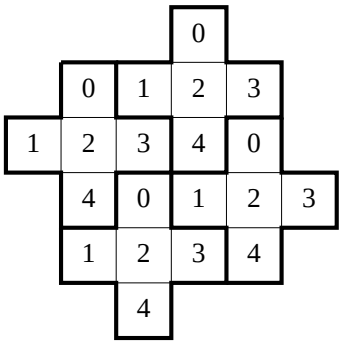

Figure 9: *Type 1 tiling groups.*

0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8
0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8

Figure 10: *Type 2 tiling groups.*

NPSF's, operations are applied to all cells having the same number. For the type 1 and type 2 neighborhoods two keywords '*type1*' and '*type2*' are defined that can be specified as the '*tiling groups type*' instead of the normal '*tile size*'.

$$\langle \textit{tile size} \rangle_{\textit{tiling groups}} \quad ::= \quad \underline{[}\langle \textit{tiling groups type} \rangle\underline{]} \tag{63}$$

$$\begin{aligned} \langle \textit{tiling groups type} \rangle \quad ::= \quad & \textit{type1} \mid \\ & \textit{type2} \mid \\ & \ldots \end{aligned} \tag{64}$$

Now the '*location*' of a local operation consists of only one '*index*', specifying the cell number within the tiling group.

$$\langle \textit{location} \rangle \quad ::= \quad \underline{[}\textit{index}\underline{]} \tag{65}$$

$$\langle \textit{index} \rangle \quad ::= \quad \langle \textit{integer} \rangle \tag{66}$$

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*'.

$$\langle \textit{tiling group type} \rangle = \langle \textit{tile size} \rangle \langle \textit{location numbering} \rangle \tag{67}$$

The '*location numbering*' consists of a list, between parentheses '(...)', of '*integers*', separated by comma's ',', specifying the location numbers within the tile from left to right and from top to bottom (actually, both in positive direction).

$$\langle \textit{location numbering} \rangle \quad ::= \quad \underline{(}\langle \textit{integer} \rangle \{ \underline{,} \langle \textit{integer} \rangle \} \underline{)} \tag{68}$$

Although tiling groups can be defined within the test, tiling group type definitions will often precede the actual test.

The type 2 tiling groups as depicted in figure 10 are defined as a one-to-one match:

$$\textit{type2} = [3,3](0,1,2,\ 3,4,5,\ 6,7,8)$$

To define the type 1 tiling groups as depicted in figure 9, a 5×5 tile is required, as can easily be seen from figure 11. The definition of the type 1 tiling groups can be specified as:

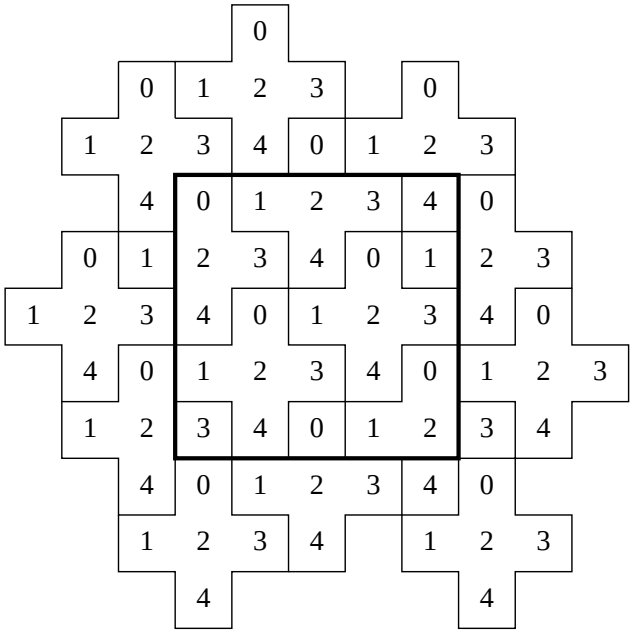

Figure 11: *Tile for type 1 tiling groups.*

$$\textit{type1} = [5,5](0,1,2,3,4,\ 2,3,4,0,1,\ 4,0,1,2,3,\ 1,2,3,4,0,\ 3,4,0,1,2)$$

The checkerboard pattern as depicted in figure 7 would be specified as:

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

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*'

for notations involving pseudo random memory tests (see section 4.7).
In this section '*line operations*' are introduced [Inoue, 1987 and Matsuda, 1989]. These operations act on complete rows in the memory cell array.

$$\langle special\ operation\rangle \quad ::= \quad \langle line\ operation\rangle \mid \ldots \tag{69}$$

A read '*line operation*' ('$_{line}r$') reads all bits in the current row, using an '*interleaving*' and '*offset*'. The '*interleaving*' defaults to 1, the '*offset*' to 0. 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, again using an '*interleaving*' and an '*offset*' with the same defaults.

$$\langle line\ operation\rangle \quad ::= \quad {}_{line}r^{\langle interleaving\rangle,\langle offset\rangle}\langle expected\ data\rangle \mid {}_{line}w^{\langle interleaving\rangle,\langle offset\rangle}\langle data\rangle \tag{70}$$

$$\langle interleaving\rangle \quad ::= \quad \langle integer\rangle \tag{71}$$

$$\langle offset\rangle \quad ::= \quad \langle integer\rangle \tag{72}$$

$$\langle expected\ data\rangle_{line\ operation} \quad ::= \quad \langle binary\ digit\rangle \mid \langle empty\rangle \tag{73}$$

$$\langle data\rangle_{line\ operation} \quad ::= \quad \langle binary\ digit\rangle \tag{74}$$

For example, the following march element writes and reads 1's to and from the memory cell array, accessing complete lines:

$$\updownarrow ({}_{line}w1,{}_{line}r1)$$

Another example, now writing and reading 0's to and from a memory cell array filled with 1's, using an interleaving of 4:

$$\updownarrow ({}_{line}w1);\updownarrow ({}_{line}w^{4,0}0,{}_{line}r^{4,0}0,{}_{line}w^{4,0}1,{}_{line}w^{4,1}0,{}_{line}r^{4,1}0,{}_{line}w^{4,1}1,{}_{line}w^{4,2}0,{}_{line}r^{4,2}0,{}_{line}w^{4,2}1,$$
$${}_{line}w^{4,3}0,{}_{line}r^{4,3}0,{}_{line}w^{4,3}1)$$

4.7 Pseudo random tests

Pseudo random tests are a completely different class of memory tests. Most notable is that the test specification is depending on the number of rows and columns of the memory cell array (i.e. the size of the address decoders) of the memory under test. None of the tests we have seen previously has this dependance (although it is possible to specify non-relative borders in the expressions of the addressing ranges).

A '*pseudo random march element*' consists of an addressing, followed by '*pseudo random operations*', separated by comma's ',', between parentheses '(...)', or of a '*pseudo random crc operation*'.

$$\langle pseudo\ random\ march\ element\rangle ::= \tag{75}$$
$$\quad {}_{pseudo\ random}\mathcal{A}^{\langle addressing\ sequence\rangle}_{\langle addressing\ range\rangle}(\langle pseudo\ random\ operation\rangle\{,\langle pseudo\ random\ operation\rangle\}) \mid$$
$$\quad \langle pseudo\ random\ crc\ operation\rangle$$

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*'.

$$\langle addressing\ sequence\rangle_{pseudo\ random\ march\ element} ::= \tag{76}$$
$$\quad \langle pseuso\ random\ addressing\ sequence\rangle;\langle pseudo\ random\ generator\rangle$$

A '*pseudo random addressing sequence*' can be an ordinary '*1 dimensional pseudo random addressing sequence*' or '*2 dimensional pseudo random addressing sequence*'. These have 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.

$$\langle \textit{pseudo random addressing sequence} \rangle ::= \langle \textit{1 dimensional pseudo random addressing sequence} \rangle \mid \langle \textit{2 dimensional pseudo random addressing sequence} \rangle \mid \langle \textit{pseudo random pseudo random addressing sequence} \rangle \quad (77)$$

$$\langle \textit{1 dimensional pseudo random addressing sequence} \rangle ::= \langle \textit{direction} \rangle \textit{mca} \quad (78)$$

$$\langle \textit{2 dimensional pseudo random addressing sequence} \rangle ::= \langle \textit{direction} \rangle \textit{row}_{\underline{,}} \langle \textit{direction} \rangle \textit{column} \mid \langle \textit{direction} \rangle \textit{column}_{\underline{,}} \langle \textit{direction} \rangle \textit{row} \mid \langle \textit{direction} \rangle \textit{row} \mid \langle \textit{direction} \rangle \textit{column} \quad (79)$$

$$\langle \textit{pseudo random pseudo random addressing sequence} \rangle ::= \langle \textit{pseudo random pattern} \rangle \quad (80)$$

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. It consists of '*pseudo random pattern elements*', these being '*pseudo random words*' and '*pseudo random digits*'.

$$\langle \textit{pseudo random pattern} \rangle ::= \langle \textit{pseudo random pattern element} \rangle \{ \langle \textit{pseudo random pattern element} \rangle \} \quad (81)$$

$$\langle \textit{pseudo random pattern element} \rangle ::= \langle \textit{pseudo random word} \rangle \mid \langle \textit{pseudo random digit} \rangle \quad (82)$$

A '*pseudo random word*' can be a part taken from the pseudo random string, specified by a question mark '?' subscripted with the '*begin index*' and '*end index*' of the desired pseudo random substring (the lesb is number 0). Or it can be just an ordinary '*binary word*'.

$$\langle \textit{pseudo random word} \rangle ::= ?_{\langle \textit{begin index} \rangle_{\underline{,}} \langle \textit{end index} \rangle} \mid \langle \textit{binary word} \rangle \quad (83)$$

$$\langle \textit{begin index} \rangle ::= \langle \textit{integer} \rangle \quad (84)$$

$$\langle \textit{end index} \rangle ::= \langle \textit{integer} \rangle \quad (85)$$

A '*pseudo random digit*' can be a digit taken from the pseudo random string, specified by a question mark '?' subscripted with the '*index*' of the desired digit in the pseudo random string. Or it can be a simple '*binary digit*'.

$$\langle \textit{pseudo random digit} \rangle ::= ?_{\langle \textit{index} \rangle} \mid \langle \textit{binary digit} \rangle \quad (86)$$

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. Swapping the indices, i.e. $?_{1,4}$, selects the same substring, but mirrors it, here resulting in 1010.

An example combining with binary digits and overlapping selections: the pseudo random pattern $11?_400?_{6,4}$ 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*', that specifies the next pseudo random string from the current one.

$$\langle \textit{pseudo random generator} \rangle \quad ::= \quad \langle \textit{seed} \rangle , \langle \textit{characteristic polynomial} \rangle \tag{87}$$

$$\langle \textit{seed} \rangle \quad ::= \quad \langle \textit{binary word} \rangle \tag{88}$$

The '*characteristic polynomial*' is not further specified; often it will be something like '$1 + x + x^2 + x^4$'.

$$\langle \textit{characteristic polynomial} \rangle \quad ::= \quad \ldots \tag{89}$$

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

$1000, x + x^3$

The '*addressing range*' of a pseudo random march element is always empty.

$$\langle \textit{addressing range} \rangle_{\textit{pseudo random march element}} \quad ::= \quad \langle \textit{empty} \rangle \tag{90}$$

A '*pseudo random operation*' can be a '*r*' (read), possibly followed by a '*pseudo random pattern*' (mostly not) specifying the expected data, or a '*w*' (write) followed by a '*pseudo random pattern*' specifying the to be written data. Or a '*pseudo random operator*', in which case the operation itself (a read or a write operation) is pseudo random. The following '*pseudo random pattern*' is ignored in case the pseudo random operator turns out to be a read operation. The last pseudo random operation is the '*pseudo random crc operation*', which control the crc logic often used with pseudo random BIST's (Build-In Self Tests).

$$\begin{aligned} \langle \textit{pseudo random operation} \rangle \quad ::= \quad & r\langle \textit{pseudo random pattern} \rangle \mid \\ & r \mid \\ & w\langle \textit{pseudo random pattern} \rangle \mid \\ & \langle \textit{pseudo random operator} \rangle \langle \textit{pseudo random pattern} \rangle \mid \\ & \langle \textit{pseudo random crc operation} \rangle \end{aligned} \tag{91}$$

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.

$$\langle \textit{pseudo random operator} \rangle \quad ::= \quad \frac{r}{w}_{\langle index \rangle} \tag{92}$$

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:

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

Another example, using the pseudo random operator:

${}_{\textit{pseudo random}}\mathcal{A}^{+mca;1010,1+x^3}(w?_{2,1}, \frac{r}{w}_2 ?_{3,2}, r)$

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 signture is compared against the value known to be correct. A '*pseudo random crc operation*' can be a '*pseudo random crc reset operation*', notated as '*crc R*', that resets the pseudo random crc counter. Or a '*pseudo random crc check operation*' that consists of the keyword '*crc*' followed by the '*expected data*'.

$$\begin{aligned} \langle \textit{pseudo random crc operation} \rangle \quad ::= \quad & \langle \textit{pseudo random crc reset operation} \rangle \mid \\ & \langle \textit{pseudo random crc check operation} \rangle \end{aligned} \tag{93}$$

$$\langle \textit{pseudo random crc reset operation} \rangle \quad ::= \quad \textit{crc R} \tag{94}$$

$$\langle \textit{pseudo random crc check operation} \rangle \quad ::= \quad \textit{crc}\langle \textit{expected data} \rangle \tag{95}$$

So, a typical CRC BIST will have the following pattern:

$crcR, \ldots, crc$

4.8 Global operations

'*global march elements*' contain '*global operations*' that affect the entire memory.

$$\langle \textit{global march element} \rangle \quad ::= \quad \langle \textit{global operation} \rangle \tag{96}$$

A '*global operation*' can be a '*reset operation*', a '*delay operation*', a '*clock operation*', a '*Vcc operation*', or a '*temperature operation*'.

$$\begin{aligned} \langle \textit{global operation} \rangle \quad ::= \quad & \langle \textit{reset operation} \rangle \mid \\ & \langle \textit{delay operation} \rangle \mid \\ & \langle \textit{clock operation} \rangle \mid \\ & \langle \textit{Vcc operation} \rangle \mid \\ & \langle \textit{temperature operation} \rangle \end{aligned} \tag{97}$$

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

$$\langle \textit{reset operation} \rangle \quad ::= \quad R \tag{98}$$

Some faults in the memory may take time to develop, e.g. data retention faults [Dekker, 1988]. Tests to detect these require time to elapse without read or write operations being applied to the memory. Until now each operation took exactly one cycle. A 'd' operation can be used to specify the elapse of one or more (as specified by '*times*') time cycles. Using the 'D' operation followed by a '*time*' specification, a certain amount of time can be passed. The exact notation of '*time*' is not further specified; for example, it may be notated like '$250ms$'. A 'D' operation without any time specification will cause enough time to pass as needed for all effects to become extinct.

$$\begin{aligned} \langle \textit{delay operation} \rangle \quad ::= \quad & \langle \textit{times} \rangle d \mid \\ & D \langle \textit{time} \rangle \end{aligned} \tag{99}$$

$$\begin{aligned} \langle \textit{time} \rangle \quad ::= \quad & \ldots \mid \\ & \langle \textit{empty} \rangle \end{aligned} \tag{100}$$

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*', consisting of a 'C' followed by the '*clock speed*' itself. The exact notation of the '*clock speed*' is not further specified. Mostly, it will just consist of a real value followed by a 'MHz' sign, i.e. '$100MHz$'. If no '*clock speed*' is specified explicitly, the clock speed will be reset to its normal value.

$$\langle \textit{clock operation} \rangle \quad ::= \quad C \langle \textit{clock speed} \rangle \tag{101}$$

$$\begin{aligned} \langle \textit{clock speed} \rangle \quad ::= \quad & \ldots \mid \\ & \langle \textit{empty} \rangle \end{aligned} \tag{102}$$

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}. V_{cc} is set to the value following the 'Vcc' keyword. The exact notation of 'Vcc' is not

further specified. Mostly, it will just consist of a real value followed by a 'V' sign, i.e. '$2.0V$'. If no 'Vcc' is specified explicitly, V_{cc} will be reset to its normal value.

$$\langle Vcc\ operation\rangle \quad ::= \quad Vcc\langle Vcc\rangle \tag{103}$$

$$\langle Vcc\rangle \quad ::= \quad \ldots \mid \langle empty\rangle \tag{104}$$

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*'. The temperature is set to the value following the 'T' keyword. The exact notation of the '*temperature*' is not further specified. Mostly, it will consist of a real value followed by a 'K' (Kelvin), 'C' (Celsius) or 'F' (Fahrenheit) sign, i.e. '$70C$'. If no '*temperature*' is specified explicitly, the operating temperature will be reset to its normal value.

$$\langle temperature\ operation\rangle \quad ::= \quad T\langle temperature\rangle \tag{105}$$

$$\langle temperature\rangle \quad ::= \quad \ldots \mid \langle empty\rangle \tag{106}$$

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 within the curly braces '$\{\ldots\}$'.

$$\langle sequential\ operator\rangle \quad ::= \quad \langle variable\ assignment\rangle \mathcal{S}_{begin\ running\ index}^{end\ running\ index} \langle expression\rangle \{,\langle expression\rangle\} \underline{\{\ldots\}} \tag{107}$$

$$\langle variable\ assignment\rangle \quad ::= \quad \langle variable\rangle = \tag{108}$$

$$\langle variable\rangle \quad ::= \quad \alpha \mid \beta \mid \ldots \omega \tag{109}$$

Most of the times, a simpler notation can be used. For example, the assignment of the values 0 and 1 to the variable δ can be specified as: $\mathcal{S}_{\delta=0,1}\underline{\{\ldots\}}$.

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. This results 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.

$$\langle parallel\ operator\rangle \quad ::= \quad \langle variable\ assignment\rangle \mathcal{P}_{begin\ running\ index}^{end\ running\ index} \langle expression\rangle \{,\langle expression\rangle\} \underline{\{\ldots\}} \tag{110}$$

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.

- *MATS+* [Nair, 1979 and Abadir, 1983]:

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

  In a memory cell array containing all 0's a 1 is written to each cell after checking its contents. Then in reverse order a 0 is written into each cell after verifying that it contains a 1.

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

  $\{\Updownarrow (w0); \nearrow (\nwarrow (w1), \Updownarrow (r), \nwarrow (w0));$
  $\Updownarrow (w1) : \nearrow (\nwarrow (w0), \Updownarrow (r), \nwarrow (w1))\}$

  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:

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

- The *GALROW* test [Breuer, 1976] does the same thing as Walking 1/0, except that it reads the base cell after each read operation from its row:

  $\{\Updownarrow (w0); \Updownarrow_a (w1, \leftrightarrow_{\neg a} (r0, r_a 1), w0);$
  $\Updownarrow (w1); \Updownarrow_a (w0, \leftrightarrow_{\neg a} (r1, r_a 0), w1)\}$

- The *moving inversions* addressing sequence used with the MOVI test [de Jonge, 1976] requires the introduction of a special addressing sequence. The '*pmovi*' '*sequence*' (partial MOVI) specifies an addressing sequence increasing at the index specified by the '*integer*' following the '*pmovi*' keyword, using end-around carry. If the '*pmovi*' march elements are sub march elements of a march element with a '*movi*' '*sequence*', which generates all address indices for the partial moving inversions, the index '*integer*' following the '*pmovi*' keyword is omitted.

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

- *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 *Hammer* test issues a base cell set to 1, walking over the diagonal of the memory cell array containing all 0's. Each base cell is written 1000 times. Then the row and column of the base cell, and the base cell itself, are checked, before continuing to the next base cell. After completion the test is repeated using a base cell with value 0 in a memory cell array containing all 1's.

  $\{\Updownarrow(w0); \swarrow\!\!\nearrow_a (1000w1, \leftrightarrow_{\neg a}(r0), r1, \updownarrow_{\neg b}(r0), r1, w0);$
  $\Updownarrow(w1); \swarrow\!\!\nearrow_a (1000w0, \leftrightarrow_{\neg a}(r1), r0, \updownarrow_{\neg b}(r1), r0, w1)\}$

- A *multi-port* 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 has just been written.

- 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:

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

- A *MAMB* test (Multiple-Arrays, Multiple-Bits, [van de Goor, 1991]):

  $\mathcal{P}_{array}\{\updownarrow ({}_{line}w^{2,0}0, {}_{line}w^{2,1}1); \updownarrow ({}_{line}r^{2,0}0, {}_{line}r^{2,1}1);$
  $\updownarrow ({}_{line}w^{2,0}1, {}_{line}w^{2,1}0); \updownarrow ({}_{line}r^{2,0}1, {}_{line}r^{2,1}0)\}$

  For each array in parallel, a pattern of alternating 0's and 1's is written and read using line mode access. Then the same test is repeated with the 0's and 1's swapped.

- *March G* [van de Goor, 1993]:

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

- A *data retention* test:

  $\{\Updownarrow(w0); V_{cc}2.0V; D250ms; V_{cc}; \Updownarrow(r0);$
  $\Updownarrow(w1); V_{cc}2.0V; D250ms; V_{cc}; \Updownarrow(r1)\}$

  This test sets the complete memory cell array to 0, lowers V_{cc} to $2.0V$ for 250 ms, and then checks all cells. The test is repeated for a background of all 1's.

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

[16] 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.

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