Generic, Orthogonal and Low-cost March Element based Memory BIST

Ad J. van de Goor¹,²
¹ComTex
Voorwillenseweg 201
2807 CA Gouda, The Netherlands
Ad.vd.Goor@kpnplanet.nl

Said Hamdioui²
²Delft University of Technology
Faculty of EE, Mathematics and CS
Mekelweg 4, 2628 CD Delft, The Netherlands
S.Hamdioui@tudelft.nl

Abstract

This paper contributes to the field of MBIST architecture and implementation by addressing the two most area-critical components: the Command Memory (ComMem) and the Address Generator (AddrGen). The ComMem area is minimized by using a novel MBIST architecture, based on the Generic March Element (GME) concept. A GME is a March Element which specifies the required operations and the generic data values; it can be specified independent of the algorithm stresses. The AddrGen area is minimized by using an efficient implementation, based on a single Up-counter and a set of multiplexors. The experimental results show that the proposed MBIST outperforms the existing MBISTs in terms of area, power, speed, and flexibility. E.g., for a 16Kx16-bit memory, the proposed MBIST consumes about 40% less area and operates at least 1.6 times faster than the state-of-the-art.

Keywords: Memory Testing, MBIST, Generic March Elements, Orthogonality, Address Generators.

I. Introduction

Memory Built-In Self-Test (MBIST) is a common industrial practice for testing the large number of embedded memories in a System-on-chip. This important topic has been addressed by many authors [1]-[2]. When implementing MBIST engines, tradeoffs are made on: (a) the number of supported algorithms, (b) the flexibility of the MBIST engine (in order to cope with the unexpected), (c) the implementation speed, and (d) the area overhead.

When MBIST performs tests, memory accesses have to be done at-speed using Back-to-Back (BtB) memory cycles [4] - [9]. Systems require large, high speed memories, while the technology scaling exhibits a large spread in implementation parameters, resulting in speed-related (delay) faults [5], [8], [11], [12]. Their detection requires non-linear algorithms (such as GalPat, GalRow and GalColumn) and special stresses [4], [3], [12], [17]. However, the implementation for such MBISTs is not that easy; one typically has to exploit expensive prefetching and pipelining techniques to satisfy the BtB cycle requirement [3], [6]. Therefore, an architecture which can support a large variety of algorithms (linear and non-linear), and a large variety of stresses (e.g., different address sequences), while allowing for a simple implementation, is the key to a flexible and low-cost MBIST.

Figure 1 shows the relative area -in % - taken by the main MBIST components for three MBIST designs [3], [16], [13]. These components consist of Control (Ctrl), test algorithm Command Memory (ComMem), Instruction fetch and decode (Instr), Address Generator (AddrGen) and the Data Generator (DataGen). The figure clearly shows that two components take up most of the area: the ComMem consumes about 39%, and the AddrGen about 28%. Therefore, optimizing their area will significantly reduce the overall area; this is the target of this paper.

This paper proposes a new MBIST engine which reduces the ComMem area and provides higher flexibility and at-speed testing, by using a novel architecture, based on the Generic March Element (GME) concept. The GME is a March Element (ME) primitive which specifies the required operations and the generic data values. This is orthogonal to all other stresses, such as the address order, the address direction, the counting method, etc. It will be shown that the set of GMEs, required to compose most industrial algorithms, is small; this reduces the size of the...
ComMem. In addition, the AddrGen area is minimized through the use of a single Up-counter, which, together with a set ofmuxes, can generate the required address sequences. The experimental results show that the proposed MBIST outperforms the existing MBISTs in terms of speed and area overhead. E.g., for a 16K x 16-bit memory, the area overhead is 7% and the speed is 500MHz; this is 40% less area overhead and 50% faster than the state-of-the-art [13].

The outline of this paper is as follows. Section 2 provides the list of algorithms of interest and their stresses. Section 3 gives an analysis of the algorithms and shows that they can be composed of only a few different GMEs. Section 4 proposes the GME-MBIST engine architecture to minimize the ComMem. Section 5 shows the minimal AddrGen implementation. Section 6 presents the simulation results and compares them with the existing MBIST designs, while Section 7 ends with conclusions.

II. Memory algorithms and their specification

This section introduces the notation for the algorithms and stresses, and gives the list of algorithms of interest to the MBIST engine.

A. Algorithm notation

The algorithms in this paper consist of linear and non-linear algorithms, described with an extended notation for march algorithms. March algorithms are the most common linear algorithms, described with an extended notation for them. March algorithms stresses are of interest to the Fault Coverage (FC) of the algorithm. Algorithm stresses are the way the algorithm is performed, and therefore influences the sequence and/or the type of the memory operations. Stress is important for the Fault Coverage (FC) of the algorithm [17]-[20]. The following algorithm stresses are of interest for this paper:

1) The Address Direction (AD): It specifies the direction of the address sequence, which can be: Fast-row, Fast-column and Fast-diagonal, which increments or decrements the row address (column address, diagonal address) most frequently. It is specified with the subscripts r, c and d of the AO; e.g., r↑ indicates ↑ AO with the Fast-row AD.

2) The Counting Method (CM): It determines the address sequence. It has been shown that the CM is important for detecting Address Decoder Delay Faults (ADDFs) [2], [8], [22]. The most common CM is the Linear CM, denoted by the superscript ‘L’ of the AO (e.g., L↑), where L specifies the address sequence 0,1,2,3, etc. Because it is the default CM, the superscript ‘L’ is usually deleted. Another CM is the Address Complement (Ac) CM. It specifies an address sequence: 000, 111, 001, 110.
### TABLE I. Set of candidate algorithms

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>BRGM#(e)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scan</td>
<td>0(0,1)</td>
<td>({w(0,0); {r(0,0); {w(1,1); {r(1,1)}}}})</td>
</tr>
<tr>
<td>4</td>
<td>MATIS+</td>
<td>0(0,2)</td>
<td>({w(0,0); {r(0,0); {r(1,0)}}})</td>
</tr>
<tr>
<td>3</td>
<td>March C-</td>
<td>0(0,1,2)</td>
<td>({w(0,0); {r(0,0); {r(1,0)}}})</td>
</tr>
<tr>
<td>4</td>
<td>March C+</td>
<td>0(0,1,3)</td>
<td>({w(0,0); {r(0,0); {r(1,0)}}; {r(0,1,0)}})</td>
</tr>
<tr>
<td>5</td>
<td>PREV</td>
<td>0(0,3)</td>
<td>({w(0,0); {r(0,0); {r(1,0)}; {r(0,1,1)}})</td>
</tr>
<tr>
<td>6</td>
<td>March B</td>
<td>0(0,4,5,6)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}; {r(0,1,0)}})</td>
</tr>
<tr>
<td>7</td>
<td>Alg. B</td>
<td>0(0,6,7)</td>
<td>({w(0,0); {r(0,0); {r(1,0,0)}})</td>
</tr>
<tr>
<td>8</td>
<td>March G</td>
<td>0(0,3,4,5,6)</td>
<td>({w(0,0); {r(0,0); {r(1,0,0)}; {r(0,1,0)}})</td>
</tr>
<tr>
<td>9</td>
<td>March U</td>
<td>0(0,2,7)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}; {r(0,1,0)}})</td>
</tr>
<tr>
<td>10</td>
<td>March LR</td>
<td>0(0,1,4,7)</td>
<td>({w(0,0); {r(0,0); {r(1,0,0)}; {r(1,0,1)}})</td>
</tr>
<tr>
<td>11</td>
<td>March LA</td>
<td>0(0,1,8)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}; {r(1,0,0)}})</td>
</tr>
<tr>
<td>12</td>
<td>March NN</td>
<td>0(0,8)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}; {r(1,0,0)}})</td>
</tr>
<tr>
<td>13</td>
<td>March RAW</td>
<td>0(0,1,10)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}; {r(1,0,0)}})</td>
</tr>
<tr>
<td>14</td>
<td>March RK</td>
<td>0(0,1,11)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}; {r(1,0,0)}})</td>
</tr>
<tr>
<td>15</td>
<td>BLBF</td>
<td>0(0,1)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>16</td>
<td>dADF-Ra-WAC</td>
<td>0(0,2)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>17</td>
<td>GalPot</td>
<td>0(0,2)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>18</td>
<td>GalRow</td>
<td>0(0,3)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>19</td>
<td>GalCol</td>
<td>0(0,3)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>20</td>
<td>Gal2R</td>
<td>0(0,14)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>24</td>
<td>Butterfly</td>
<td>0(10)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>26</td>
<td>Walk10</td>
<td>0(10)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>27</td>
<td>WalkCol</td>
<td>0(10)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>29</td>
<td>HamWh</td>
<td>0(0,5)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>29</td>
<td>HamRh</td>
<td>0(0,8)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>30</td>
<td>HamDWhc</td>
<td>0(10)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>30</td>
<td>HamDWhc</td>
<td>0(10)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>29</td>
<td>MOV1</td>
<td>0(0,3)</td>
<td>({w(0,0); {r(0,0); {r(1,0,1)}})</td>
</tr>
<tr>
<td>30</td>
<td>DELAY</td>
<td>0(0,2,14)</td>
<td>({w(0,0); {r(0,0); {r(1,0,0)}})</td>
</tr>
</tbody>
</table>

### III. Analysis of the algorithms

Inspecting Table I reveals that many algorithms use the same March Elements (MEs); they may only differ in the address order and/or the data value. E.g., MATIS+ (Alg #2): \(\{w(0,0); \{r(0,0); \{r(1,0,0)\}\}\) |

March C- (Alg #3): \(\{w(0,0); \{r(0,0); \{r(1,0,0)\}\}\) |

The first and second MEs of both tests are the same; the third ME of MATIS+ is the same as that of March C-, except that the address orders are different. In addition, all MEs of March C- have the form of \(\{w(0,0); \{r(0,0); \{r(1,0,0)\}\}\) where the data value ‘D’ can be either 0 or 1, and \(\{1\} = \{0\}\), except the first and the last MEs. Hence, March C- can be implemented with three Generic March Elements (GMEs): \(\{w(0,0); \{r(0,0); \{r(1,0,0)\}\}\) |

Note that a GME is a March element only specifying the operations and their generic data values; it is orthogonal to all other specifications, such as the address order, etc.

Table II lists the set of GMEs, required for the support of the algorithms of Table I [18]. Because the # of GMEs is more than 16, the set of GMEs is divided into banks, each with up to 16 GMEs. The number 16 is dictated by the fact that the MBIST engine is controlled by commands having a length which is a multiple of 4 bits, while a 4-bit field in the commands is used to specify the GME. In this natural text, the following abbreviations are used: 010, 011, and 100 [10], [8]; each bold address is the 1’s compliment of the preceding address.

The 2’s CM is yet another CM: typically used by the MOV1 algorithm [10], [8]. It repeats the PMOVI algorithm N times (\(N = \) the number of memory address bits) with an address increment/decrement value of 2\(^i\); with 0 \(\leq i \leq N - 1\).

3) The Data Background (DB): It is the data pattern which is actually in the cells of the memory cell array. The four DBs commonly used in industry are: Solid (sDB): all 0’s (i.e., 0000...0000...) or all 1’s. Checkerboard (bDB): 0101...0101.../1010... Column Stripes (cDB): 0101.../0101.../0101... Row Stripes (rDB): 0000.../1111.../0000.../1111...

A ‘w0’ means that the selected DB is applied; a ‘w1’ means an inverse of that DB is applied.

### Paper 8.1 INTERNATIONAL TEST CONFERENCE
TABLE II. Generic March Elements

<table>
<thead>
<tr>
<th>B</th>
<th>GME#</th>
<th>GME Description</th>
<th>Alg. #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>↑(wD)</td>
<td>1-30</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>↑(rD)</td>
<td>1,3-4,10-13</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>↑(rD, wD)</td>
<td>2,3-9,10,16,30</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>↑(rD, wD, rD)</td>
<td>4,5,8,29</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>↑(rD, wD, rD, wD, rD, wD)</td>
<td>6-8</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>↓(rD)</td>
<td>6,8</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>↓(rD, wD)</td>
<td>6-8</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>↓(rD, wD, rD, wD)</td>
<td>7-9,10,14</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>↓(rD, wD, wD, rD)</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>↓(rD, rD, wD, rD, wD)</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>↓(rD, wD, rD, rD, wD, rD, wD)</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>↓(wD, rD)</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>↓(wD, rD, wD, rD)</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>↓(rD, rD)</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>Del50</td>
<td>30</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>Del100</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>↑(wD)</td>
<td>1-30</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>↑(rD)</td>
<td>1,3-4,10-13</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>↑(wD, wD, rD, rD, wD, wD)</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>↑(wD, wD, wD, X)</td>
<td>18-19</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>↑(wD, wD, wD, rD, rD, wD)</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>↑(wD, wD, wD, rD, rD, wD)</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>↑(rD, wD, wD, X)</td>
<td>23-24</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>↑(rD, wD, rD)</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>↑(rD, wD, rD, wD, rD)</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>↑(wD, wD, wD, rD)</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>↑(wD, wD, wD, rD)</td>
<td>28</td>
</tr>
</tbody>
</table>

x ∈ {R, C}; Data Value D ∈ (0, 1); Del = inverse of D

proposed the GMEs of most linear algorithms are placed in Bank 0, while the GMEs of the hammer and the non-linear algorithms are placed in Bank 1. Conceptually, the number of banks can be arbitrary large as will be explained with `load bank` instruction. The first 2 GMEs of both banks are made in the same order to reduce the amount of bank-switching, for which a special command is available; see Section IV-B. The first column 'B' of Table II lists the Bank, the second column the Generic March Element # (GME#); each GME is assigned a unique # in a bank. Last, the column 'Alg#' lists the algorithms of Table I which use the corresponding GME#. The column B(GME#) in Table I shows which Bank 'B' and GMEs of Table II are used for the corresponding algorithm. For example, PMOV1 (i.e., Alg#5) is composed of GME#0 and GME#3 of Bank 0 denoted as 0,0,3. Table II shows that only a small number of GMEs are required to support the algorithms of Table I. Many commercial sets of algorithms would require less than 16 GMEs.

IV. The GME-MBIST engine architecture

Figure 2 shows a high-level block diagram of the GME-MBIST engine. The 'Memory under test' is controlled by a set of orthogonal signals, which can be combined in any way. For example, the GME is orthogonal to any of the components of the Stress Combination (SC), for example the AO and the AD; this allows for the application of any GME with any SC.

The architecture of the GME-MBIST engine will be described in terms of its registers and its commands. They are used to implement the tests, as shown in the examples of Section 4.3. Throughout the remainder of this section, the basic architecture will be covered together with a set of extensions, which illustrate the ease with which new features and capabilities can be added to the basic architecture.

A. The GME-MBIST registers

The GME-MBIST architecture supports a set of Basic registers, which are part of the minimal GME-MBIST engine, while the Extension registers support some of the advanced features of the GME-MBIST engine. The register naming convention is such that the register name includes its size. For example, 'CC[5..0]' denotes the 6-bit Command Counter, its most significant bit is bit-5, its least significant bit is bit-0; 'AOR[0]' denotes the 0-bit Address Order Register. Table III summarizes the description of the register set of the GME-MBIST engine.

Basic registers

The basic registers are part of the minimal GME-MBIST engine; their presence is mandatory [18], [13].

1) CC[5..0]: Command Counter. Size depends on Command Memory (ComMem) size; default: 6 bits. The CC points to a location in the ComMem, which contains the to-be-executed command.

2) CM[63..0,3..0]: Command Memory (ComMem). Size depends on the used test set; default: 64x4-bit nibbles. The ComMem contains the commands which specify the to-be-executed tests.

3) CR[7..0]: Command Register. Contains the first 2 nibbles of the command.

4) AOR[0]: Address Order Register. Specifies the Data
TABLE III. GME-MBIST engine registers

<table>
<thead>
<tr>
<th>Reg. Name</th>
<th>BE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC[5..0]</td>
<td>B</td>
<td>Command Counter</td>
</tr>
<tr>
<td>CM[63..0,3..0]</td>
<td>B</td>
<td>Command Memory</td>
</tr>
<tr>
<td>CR[7..0]</td>
<td>B</td>
<td>Command Register</td>
</tr>
<tr>
<td>ADR[7..0]</td>
<td>B</td>
<td>Address Order Register</td>
</tr>
<tr>
<td>DVR[0]</td>
<td>B</td>
<td>Data Value Register</td>
</tr>
<tr>
<td>DBR[1..0]</td>
<td>B</td>
<td>Data Background Register</td>
</tr>
<tr>
<td>GMER[3..0]</td>
<td>B</td>
<td>Generic March Element Register</td>
</tr>
<tr>
<td>BR[0]</td>
<td>E</td>
<td>Bank Register</td>
</tr>
<tr>
<td>REPR[5..0]</td>
<td>E</td>
<td>Re-execute Entry Point Register</td>
</tr>
<tr>
<td>RCNTR[3..0]</td>
<td>E</td>
<td>Re-execute Count Register</td>
</tr>
</tbody>
</table>

BE = Basic or Extension

Value (DV) used by the GME operations as follows:
0: The GME operations assume the specified DB (see DBR); 1: Use the inverted DB.

6) CMADR[1..0]: Counting Method & Address Direction Register. This register specifies the combination of the to-be-used Counting Method (CM) and the Address Direction; it can specify on of the following four options: (1) LR: Linear CM & Fast-row AD; (2) LC: Linear CM & Fast-column AD; (3) AC: Address Complement CM; note that AD is not applicable to CM; and (4) : Reserved.

7) DBR[1..0]: Data Background Register. It specifies one of the following data-backgrounds: (1) sDB: solid DB; (2) bDB: checkerboard DB; (3) rDB: row stripes DB; and (4) cDB: column stripes DB.

8) GMER[3..0]: Generic March Element Register. It is a 4-bit register and specifies the to-be-applied GME within the current bank. The GMER only contains a number, rather than a complete specification of a GME; the MBIST hardware uses the GME# to generate the appropriate operation sequence and data values.

Extension registers
The optional extension registers support additional features of the GME-MBIST architecture [18], [13].

1) BR[0]: Bank Register. It specifies the bank from which a GME has to be selected. In this paper the size of the BR is 1-bit, which allows for up to 32 GMEs. Depending on the application, it may have a different size.

2) REPR[5..0]: Re-execute Entry Point Register. The REPR is used by the REP (REPeat) and the POWi commands, which allow a Block of Commands (BoC) to be re-executed a number of times. The starting address of this BoC is stored in the REPR, which has the same size as the CC (Command Counter).

3) RCNTR[3..0]: Re-execute Count Register. The RCNTR specifies the number of times a BoC has to be re-executed by the REP command.

B. The GME-MBIST commands

The architecture supports commands to control the operations of the GME-MBIST engine. They have a variable length which is a multiple of one nibble (4 bits), such that the Command Memory size can be minimized. The most frequent commands are encoded in the smallest number of nibbles. This reflects itself in the number of bits used for specifying the Opcode; which is also variable in length. Last, similar to the registers, the set of commands can be divided into two classes: Basic and Extension.

Basic commands
The basic command set consists of only two commands; they are mandatory and allow for a minimal implementation of the GME-MBIST engine [13].

1) INIT: INITIALize. The INIT command clears GMER.
   It establishes the DB values, the CM&AD, the DV and the AO for the application of the GME# specified in the GMER (which is \( \mathbf{\uparrow} (wD) \)). Last, it establishes the Re-execute Entry Point by clearing the RCNTR and loading the start address of the to-be-re-executed Block of Commands in the REPR.

2) SGME: Select GME. This command allows for the execution of a specified GME. It requires the specification of the GME#, the DV and the AO.

Extension commands
The extension commands provide extra capability/functionality and/or reduce the required Command Memory size [13].

1) SAODV: Select AO and DV, for current GME. Several several algorithms have a sequence of MEs which only differ in the used AO and DV. For example: MATS+, March C- and PMOVI. This command specifies the AO and DV for the current GME (see Example 2 in Subsection 4.3).

2) SAODVE: Select AO, DV, CM&AD and GME#. From inspecting Table I one can conclude that the selection of the ’CM&AD’ applies to all MEs of a test. However, in some rare cases this does not hold. For example, the Philips 6n algorithm \( \{ r\mathbf{\uparrow}(w0); r\mathbf{\uparrow}(r0,w1); r\mathbf{\uparrow}(r1,w0,r0) \} \) requires the use of the SAODVE command, because the AD of the GME \( r\mathbf{\uparrow}(r1,w0,r0) \) differs from the two preceding GMEs.

3) LBR: Load Bank Register. When more than 16 GMEs are required, the set of GMEs is divided into banks with a maximum of 16 GMEs. The Bank ’B’ parameter of the LBR command specifies the bank.

4) SREP: Set Re-execute Entry Point. Sometimes MEs of a set of tests are put together in a single long
test. To save execution time, the state of the memory after a given test is used as the initial state for the following test. Then, if a part of the long test has to be re-executed, the SREP command is required to establish the Entry Point of that part.

5) REP: REPeat block of commands. Industrial test sets usually contain tests which are a repeated application of an algorithm, whereby stress conditions such as the CM&AD are varied. The REP command supports this in a very efficient way. It allows for re-executing a Block of Commands (BoC). The BoC starts at the location specified by the Re-execute Entry Point Register (REPR) and ends at the REP command. The BoC may consist of several algorithms and is repeated a number of times, as specified by the RCNTR field of the REP command (see Example 4 in Subsection 4.3).

6) POWi: Re-execute with POWer of 2^i addressing. The POWi command is a special case of the REP command: it re-executes a Block of Commands N−1 times; N is the number of address bits, which is hardwired in the GME-BIST engine. During the i-th re-execution, i is used to generate the address increment/decrement value of 2^i. The RCNTR register is used to keep track of the current value of i.

C. Examples

It is clear from the above that GME-MBIST needs only a few simple, memory-efficient commands. Its efficiency will be demonstrated with the following four examples [13].

Example 1: March C- {↑(w0); ↑(r0, w1); ↑(r1, w0); ↓(r0, w1); ↓(r1, w0); ↓(w0)}; applied with sDB and Lr. This implementation uses only the basic commands.
- ↑(w0). INIT: AO=\#, DV=0, CAMADR=Lr, DBR=bDB. Implicit application of GME#0.
- ↑(r0, w1). SGME: AO=\#, DV=0, GMER=2.
- ↑(r1, w0). SGME: AO=\#, DV=1, GMER=2.
- ↓(r0, w1). SGME: AO=\#, DV=0, GMER=2.
- ↓(r1, w0). SGME: AO=\#, DV=1, GMER=1.
- ↓(w0). SGME: AO=\#, DV=0, GMER=1.

A total of six commands are required, one per ME; they require 12 nibbles of Command Memory (ComMem) [13].

Example 2: March C- March C- will be applied with sDB and Lr, while using the extended command ‘SAODV’:
- ↑(w0). INIT: AO=\#, DV=0, CAMADR=Lr, DBR=bDB. Implicit application of GME#0.
- ↑(r0, w1). SGME: AO=\#, DV=0, GMER=2.
- ↑(r1, w0). SAODV: AO=\#, DV=1.

Note: the GME# of the previous command applies.
- ↑(r0, w1). SAODV: AO=\#, DV=0.
- ↑(r1, w0). SAODV: AO=\#, DV=1.
- ↑(w0). SGME: AO=\#, DV=0, GMER=1.

The six commands require 9 nibbles of ComMem [13]. The use of ‘SAODV’ command saves 25%, as compared with Example 1.

Example 3: GalPat: {↑(w0); ↑(w1, r1); ↑(r0, w1, r0); ↓(w0, r1); ↓(w1, r0); ↓(r0, w1)}; applied with sDB and Lr, as follows:
- LBR: B=1. Note: Bank-1 of the set of GMEs has to be selected.
- ↑(w0). INIT: AO=\#, DV=0, CMADR=Lr, DBR=bDB. Apply implicit GME#0.
- ↑(w1, r1). ↑(r0, r1, w0). SGME: AO=\#, DV=1, GMER=2.
- ↑(w1). SGME: AO=\#, DV=1, GMER=0.
- ↑(r0, w1, r0). ↓(w0, r1). SGME: AO=\#, DV=0, GMER=2.

Five commands require 10 nibbles of ComMem [13].

Example 4: REPeat(PMOVI & MATS+).

PMOVI: {↑(w0); ↑(r0, w1, r1); ↑(r1, w0, r0);
↓(r0, w1, r1); ↓(r1, w0, r0)}

MATS+: {↑(w0); ↑(r0, w1); ↓(r1, w0)}

This example demonstrates the use of the REP command applied to two tests: PMOVI & MATS+. The first time they are executed using Lr (Linear addressing & Fast-row), together with sDB (solid DB). Next, they will be re-executed for LB & dB, LB & rDB, LB & cDB, and for LC & sDB, LC & dB, LC & rDB and LC & cDB. This means that after the initial execution, they are re-executed 7 times. The following set of commands will implement the above:

PMOVI
- ↑(w0). INIT: AO=\#, DV=0, CAMADR=Lr, DBR=bDB. Apply implicit GME#0. Note that RCNT will be implicitly cleared here.
- ↑(r0, w1, r1). SGME: AO=\#, DV=0, GMER=3.
- ↑(r1, w0, r0). SAODV: AO=\#, DV=1.
- ↓(r0, w1, r1). SAODV: AO=\#, DV=0.
- ↓(r1, w0, r0). SAODV: AO=\#, DV=1.

MATS+
- ↑(w0). SGME: AO=\#, DV=0, GMER=0.
- ↑(r0, w1). SGME: AO=\#, DV=0, GMER=2.
- ↓(r1, w0). SAODV: AO=\#, DV=1.
- REPeat

REP: CNTR=8, LC-cDB, LC-rDB, LC-cDB, LC-sDB, LC-cDB, LC-rDB, LC-cDB, LC-sDB.

Nine commands require 22 nibbles of command memory for 2 tests executed 8 times [15].

V. Minimization of address generation

This section describes how the Address Generator (AddrGen) area can be minimized. This minimization is mainly based on the notion that many Counting Methods (CMs) can be derived from the Linear Up-address sequence. Figure 3 shows the concept; just by using an Up-counter and a network of Muxes, most CMs can be generated. By controlling the Mux network, one can select the appropriate CM, together with its address direction (Up or Down).

In this section, first the concept will be explored for the Linear (Li) and Address complement (Ac) AddrGens; the area and power analysis for these two AddrGens will be also presented. Then, the the Gray code (GC), the Worst-case gate delay (Wc) and the 2^i (2i) AddrGens will be discussed. Thereafter, a novel solution for the Next address
Li: Linear address CM
Ac: Address complement CM
Ge: Gray code CM
Wc: Worst case CM

LiUd: Linear AddrGen based on Up-down counter
Ac: Address complement AddrGen

A. Li and Ac AddrGens

Column 'Li' of Table IV shows the Linear address sequence for a 4-bit address \((N = 4)\), while column 'Ac' shows the Address complement address sequence.

LiUd: Linear AddrGen based on Up-down counter
Figure 4(a) shows the LiUd AddrGen using J-K flip-flops. The 'U/D' (Up/Down) control input determines whether the \(\uparrow\) or \(\downarrow\) address sequence is generated, by the 'U/D' or the 'CTRL1'. The 'U/D' control signal changes with each clock period; see Table IV. The \(Q\) output of bit \(x\) controls the muxes of all bits \(2^x\), with \(x > 0\).

LiUo: Linear AddrGen based on Up-only counter
Figure 4(b) depicts the LiUo AddrGen using an Up-only counter. The U/D control input determines whether the \(Q\) (for \(\uparrow\)) or the \(\overline{Q}\) (for \(\downarrow\)) outputs are selected. Note that a single mux, which is not in the critical signal path, is used to switch between \(\uparrow\) or \(\downarrow\) counting.

Ac: Address complement AddrGen
Column 'Ac' of Table IV shows a 4-bit address sequence for the Ac CM. The \(\text{even steps}\) (see column 'Step') of this sequence form a linear \(\uparrow\) address sequence; the addresses for the \(\text{odd steps}\), in bold font, are formed by taking the one's complement of the preceding even step. Figure 4(c) shows Ac AddrGen implementation using an Up-only counter. The 'U/D' control signal controls the most-significant address bit \(O3\), which is the least-significant counter bit '0', because \(O3\) of the Ac CM changes with each clock period; see Table IV. The \(Q\) output of bit \(x\) controls the muxes of all bits \(2^x\), with \(x > 0\).

B. Area and power analysis of Li and Ac AddrGens

The AddrGens are synthesized with the Synopsys Design Compiler [15], using the Faraday UMC 90 nm Standard Process library [14]. Table V shows the area, in terms of standard \(2\)-input NAND gates, for the the LiUd, the LiUo, the Ac, and the combined LiAc AddrGens. The col. 'Freq' lists the three operating frequencies in MHz; the columns thereafter list the area requirements for AddrGens consisting of standard 2-input NAND gates, for the the LiUd, LiUo, Ac, and combined LiAc AddrGens. The col. 'Freq' lists the three operating frequencies in MHz; the columns thereafter list the area requirements for AddrGens consisting of standard 2-input NAND gates, for the the LiUd, LiUo, Ac, and combined LiAc AddrGens. The col. 'Freq' lists the three operating frequencies in MHz; the columns thereafter list the area requirements for AddrGens consisting of standard 2-input NAND gates, for the the LiUd, LiUo, Ac, and combined LiAc AddrGens. The col. 'Freq' lists the three operating frequencies in MHz; the columns thereafter list the area requirements for AddrGens consisting of standard 2-input NAND gates, for the the LiUd, LiUo, Ac, and combined LiAc AddrGens.
The mux of bit 3 address can be derived from bit 0 of the Linear sequence, as follows: bit 0 sequence for the Gc CM. This sequence can be derived from the Linear address is ‘1’. This means that in case of the Up-only counter, the ‘0’ input of the mux will select Q3 to generate O3; see Table IV. Based on the above, the implementation of Gc AddrGen is done in an simple and easy way by using linear Up-only counter.

### C. Minimizing the Gc, Wc and 2i AddrGens

This subsection shows how the Up-only counter is used to generate optimized AddrGens for Gray code (Gc), the Worst-case Gate delay (Wc), and the 2i (2i) CMs.

#### Gc: Gray code AddrGen

The column ‘Gc’ of Table IV shows a 4-bit address sequence for the Gc CM. This sequence can be derived from the Linear sequence, as follows: bit3 of the Gc address can be derived from bit0 of the Linear address by inverting it when bit1 of the linear address is ‘1’. This is shown in Figure 6(a): the mux of bit0 is controlled by the signal ‘Q1’. Similar reasoning applies to bit1 and bit2. The mux of bit3 is controlled by the Up/Down signal, which means that in case of the Up-only counter, the ‘0’ input of the mux will select Q3 to generate O3; see Table IV. Based on the above, the implementation of Gc AddrGen is done in an simple and easy way by using linear Up-only counter.

#### Wc: Worst Case Gate Delay AddrGen

The Worst-Case Gate Delay (WCGD) algorithm [11] has been designed as a more efficient replacement of the MOVI algorithm [24]. It has the property that for each of the 2N victim addresses (v_addr), the following N address-triplets are generated: v_addr ⊕ 2j, v_addr, v_addr ⊕ 2j; for 0 ≤ j ≤ N - 1. The column ‘Wc’ of Table IV sketches part of a 4-bit Wc address sequence; i.e., for v_addr = 0000. For every v_addr of the register (Q3, Q2, Q1, and Q0), any one of the 4 address bits has to be inverted. This is accomplished in Figure 6(b) by selecting the Qj or the Qj output, under control of the corresponding mux with control input ‘2i’ j'. For example, for bit2 the mux control input is labeled ‘2i’ j’ and (2). Note that of the 4 mux control inputs only one is active, such that only one address bit is inverted.

#### 2i: The 2i AddrGen

The column ‘2^i=4’ of Table IV shows the 2i address sequence, only for address increments/decrements of 4; i.e., i=2. The 2^i CM is important for the MOVI algorithm [8], [10], which is used throughout the industry. A straightforward implementation requires a barrel shifter with N muxe, each with N inputs, to transform the Li address sequences into the 2i sequences. However, this requires a total of N+N=N^2 muxe-inputs. An economical solution is shown in Figure 6(c); it has one mux for bit0 with N inputs, and muxe with 2 inputs for all other bits. For all values of i, except for i = 0, the muxe interchange col., with colo. Therefore, the mux for bit0 requires N inputs, while the other muxe only require 2 inputs.
D. Next-address (Ne) AddrGen

Figure 6(d) shows the Next(Ne) AddrGen. The implementation is based on the idea that the increment function of the Up-only counter can be separated from the Register function. This results in two separate units: the ‘Register’ and the ‘+1 increment logic’ as shown in Figure 6(b).

E. Address generator summary

Figure 7 depicts the area required for each of the CMs covered in this paper; for the completeness, the Pseudo-Random (Pr) CM (see column ‘Pr’ in Table IV) is also included. In addition, the AddrGen capable of generating ALL considered CMs in this paper (including Pr) is also included for comparison and referred to in the figures as ALL. The figure shows that the area required by the ‘ALL’ AddrGen is 2.42 to 2.95 times the area of the Li AddrGen, depending on the size of N (the larger N, the smaller the size of the ALL AddrGen). On the other hand, the ALL AddrGen requires only 40% of the area required by a brute-force implementation; e.g., for N = 24, the ALL AddrGen requires 1054 gates, as compared with 3070 gates for the brute-force implementation [13].

VI. Experimental results and comparison

The GME MBIST hardware was synthesized with the Synopsys Design Compiler with the Faraday UMC 90 nm Standard Process library. The size of the Memory-under-Test is 16K x 16-bit, with 7-bit of row and column addresses each. The Command Memory size is 64 4-bit nibbles. It is worth noting that GME MBIST also has some additional logic to support diagnosis [13].

Figure 8 shows the required area of GME MBIST for different memory sizes with a word size of 16 bits; e.g., 16K x 16 bits [13]. The area is normalized in terms of the standard 2-input NAND gate (4µm² in 90 nm). The results are for a full implementation; i.e., support of basic and extension commands and all GMEs of Table II. The numbers on the top of each bar give the area overhead - in % - for the specified memory size. E.g., for a memory size of 1M x 16 bits, the area overhead is 0.17%. The figure shows that the MBIST size increases with the memory size, while its relative area overhead becomes negligible for larger memories. Because the proposed MBIST has a modular and flexible architecture, it enables choosing subsets of combinations of algorithms and stresses. Hence, reducing the required command memory and the overall MBIST area. E.g., if only March tests are implemented, the MBIST will consume 32% less area than the full implementation [13].

Furthermore, the simulation results show that the achievable frequency varies between 500 and 714 MHz, depending on the flexibility and memory size. A higher flexibility and a larger memory size result in a lower frequency. E.g., if only March tests are implemented, then the frequency will be 714 MHz, while this will be 500 MHz (i.e., 30% slower) for a full implementation.

Table VI gives a comparison of the major aspects between the GME MBIST and some of previous published work. As can be seen, GME MBIST outperforms the other MBISTs in terms of area (at least 40% less area), speed (at least 1.6X faster) and flexibility. Our modular and orthogonal design supports any algorithm – stress combination; it provides a higher frequency for at-speed testing, and has a higher coding efficiency resulting in an overall low area. In addition, GME MBIST is unique in supporting some stresses such as Ac and 2i counting methods.

VII. Conclusions

This paper describes a novel Memory BIST engine, based on the concept of the Generic March Element (GME) [18]. A GME specifies the sequence of operations of a March Element (ME), and is generic, because the attributes
(e.g., Address Order, Addressing Direction, etc.) still can be specified independent of the GME. The GME concept provides a flexible MBIST and results in an extremely small command memory. This makes this MBIST engine very attractive for embedded applications whereby the tests have to be stored within the MBIST engine. Moreover, all information required to support an at-speed implementation is contained in the specified GME. Hence, the required detailed information to control the memory, such as whether a read or write has to be performed, can be decoded from the GME prior to its application. Furthermore, the implementation of the address generator is optimized and only used an Up-counter with a set of multiplexors.

The simulation results and the comparison with the state-of-the-art show the superiority of the proposed MBIST in terms of area overhead, flexibility and at-speed testing. The proposed MBIST consumes an area overhead which is 40% less than the existing MBISTs, runs at least 1.6X times faster and provides unique flexibility such as supporting address complement and 2\textsuperscript{nd} counting methods.

### References


**TABLE VI. Comparison with other MBISTs**

<table>
<thead>
<tr>
<th>Method</th>
<th>Data Background</th>
<th>Counting Method</th>
<th>Command Memory</th>
<th>Total bits for March Cs</th>
<th>Technology</th>
<th>Freq (MHz)</th>
<th>AREA for 16Kx16-b mem.</th>
<th>AREA for 8Kx32-b mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested loop</td>
<td>s,cb</td>
<td>0/17 b/ced</td>
<td>43x4-b</td>
<td>160</td>
<td>0.18 µm</td>
<td>40</td>
<td>7.9 K gates</td>
<td>13.6 K gates</td>
</tr>
<tr>
<td>(4 )</td>
<td>x</td>
<td></td>
<td>8x8-b</td>
<td>216</td>
<td>0.13 µm</td>
<td>NA</td>
<td>3,33 [3]</td>
<td>6.4 K gates</td>
</tr>
<tr>
<td>(2 )</td>
<td></td>
<td></td>
<td>8x9-b</td>
<td>126</td>
<td>90 nm</td>
<td>NA</td>
<td>300</td>
<td>No diagnosis</td>
</tr>
<tr>
<td>(2 )</td>
<td></td>
<td></td>
<td>64x4-b, 16x6-b</td>
<td>36</td>
<td></td>
<td>NA</td>
<td>5.6 K gates</td>
<td>3.4 K gates</td>
</tr>
</tbody>
</table>

**Legend:**

- **µ m** indicates the area overhead
- **µ m** indicates the speed overhead
- **µ m** indicates the complexity overhead
- **µ m** indicates the power overhead

**Notes:**

- *DB Reg* indicates the designer's decision to use a DB register instead of a subset of the command memory
- *Addr. Unq.* indicates the address uniqueness
- *AC.* indicates the address complement
- *s,cb,r/c* indicates the support for address complement, read, and write control

**References**