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

Ad J. van de Goor[1,2]

[1]ComTex
Voorwillenseweg 201
2807 CA Gouda, The Netherlands
Ad.vd.Goor@kpnplanet.nl

Said Hamdioui[2] Halil Kukner[2]

[2]Delft 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.

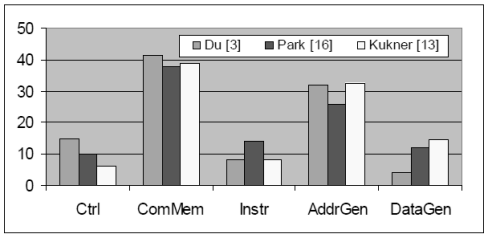

Fig. 1. AG-Overhead

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

978-1-4577-0152-8/11/$26.00

ComMem. In addition, the AddrGen area is minimized through the use of a single Up-counter, which, together with a set of muxes, 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 algorithms used for testing memories [10]. An example of a march algorithm is MATS+ [19], defined as: $\{\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0)\}$. The special symbols $\Updownarrow$, $\Uparrow$, and $\Downarrow$ are the *Address Orders (AOs)*; they determine the way one proceeds from one address to the next address. $\Uparrow$ denotes an ascending AO (e.g., 0,1,2,3,...), $\Downarrow$ denotes a descending AO, while $\Updownarrow$ denotes that the AO can be chosen freely. MATS+ consists of three *March Elements (MEs)*, which are separated by the ';' symbol. The ME e.g., '$\Uparrow(r0, w1)$' specifies the $\Uparrow$ AO, while to each address a read operation with expected value '0' will be applied, after which a '1' will be written.

B. Set of targeted algorithms

Memories can exhibit static and dynamic faults. Static faults are not time or speed dependent; their detection does not require at-speed testing. Dynamic faults are speed dependent and do require at-speed tests. Especially *Address Decoder Delay Faults* (ADDFs) [2], [8], [22] are increasing in importance, because address decoders typically consist of pre-decoders, connected to local word line decoders and local column decoders via long wires with many via's. The wiring is susceptible to resistance and capacitance variations, while the increasing number of via's experiences extra resistive defects, causing RC delays.

Table I lists a set of algorithms which can be considered candidates for a possible MBIST implementation. Other algorithms may be added if desired; nevertheless, this will not change the concept 'GME' based MBIST engine. To help the understanding, the GalPat algorithm (i.e., Alg#17) will be explained [10]. This algorithm is non-linear and has a time complexity of $O(n^2)$. It has the property that for a given victim cell (v-cell), *Address Transitions (ATs)* are made from all other memory cells, which are denoted aggressor cells (a-cells). GalPat detects all ADDFs, because each cell will become v-cell and ATs are made between all cells. The algorithm applies a *Read-after-Read (RaR)* sequence, which means that a 'rx' operation is applied to the a-cell, followed by a '$r\overline{x}$' operation to the v-cell [20], [22]. In the march notation for GalPat '$\Uparrow_v(w1_v, \Uparrow_{-v}(r0, r1_v), w0_v)$' denotes a nested ME; it is applied to each v-cell, as specified by the AO '$\Uparrow_v$'. The AO '$\Uparrow_{-v}$' means that all a-cells are visited; these are all addresses, except the v-cell, which has to be skipped; hence the subscript '-v'.

C. Algorithm stresses/MBIST requirements

In order to be able to apply the algorithms listed in Table I, the MBIST has to satisfy many requirements and support different features, either related to the operations of the algorithm itself (see Section III), or to the algorithms stresses. An algorithm stress specifies 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., $_r\Uparrow$ indicates $\Uparrow$ 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], [23]. The most common CM is the *Linear* CM, denoted by the superscript 'L' of the AO (e.g., $^L\Uparrow$), 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

#	Name	B(GME#)	Description
1	Scan	0(0,1)	$\{\Downarrow (w0); \Uparrow (r0); \Uparrow (w1); \Downarrow (r1)\}$
2	MATS+	0(0,2)	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$
3	March C-	0(0,1,2)	$\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$
4	March C+	0(0,1,3)	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0); \Downarrow (r0)\}$
5	PMOVI	0(0,3)	$\{\Downarrow (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1); \Downarrow (r1, w0, r0)\}$
6	March B	0(0,4,5,6)	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$
7	Alg. B	0(0,6,7)	$\{\Updownarrow (w0); \Uparrow (r0, w1, w0, w1); \Uparrow (r1, w0, r0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, r1, w0)\}$
8	March G	0(0,3,4, 5,6)	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0);$ $\Downarrow (r0, w1, w0); Del100; \Updownarrow (r0, w1, r1); Del100; \Updownarrow (r1, w0, r0)\}$
9	March U	0(0,2,7)	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0, w1); \Downarrow (r1, w0)\}$
10	March LR	0(0,1,2,7)	$\{\Updownarrow (w0); \Downarrow (r0, w1); \Uparrow (r1, w0, r0, w1); \Uparrow (r1, w0); \Uparrow (r0, w1, r1, w0); \Updownarrow (r0)\}$
11	March LA	0(0,1,8)	$\{\Updownarrow (w0); \Uparrow (r0, w1, w0, w1, r1); \Uparrow (r1, w0, w1, w0, r0); \Downarrow (r0, w1, w0, w1, r1); \Downarrow (r1, w0, w1, w0, r0); \Downarrow (r0)\}$
12	March SS	0(0,1,9)	$\{\Updownarrow (w0); \Uparrow (r0, r0, w0, r0, w1); \Uparrow (r1, r1, w1, r1, w0); \Downarrow (r0, r0, w0, r0, w1); \Downarrow (r1, r1, w1, r1, w0); \Updownarrow (r0)\}$
13	March RAW	0(0,1,10)	$\{\Updownarrow (w0); \Updownarrow (r0, w0, r0, r0, w1, r1); \Updownarrow (r1, w1, r1, r1, w0, r0);$ $\Updownarrow (r0, w0, r0, r0, w1, r1); \Updownarrow (r1, w1, r1, r1, w0, r0); \Updownarrow (r0)\}$
14	March SR	0(0,7,13)	$\{\Downarrow (w0); \Uparrow (r0, w1, r1, w0); \Uparrow (r0, r0); \Uparrow (w1); \Downarrow (r1, w0, r0, w1); \Uparrow (r0, r0)\}$
15	BLIF	0(0,11)	$\{\Updownarrow (w0); \Uparrow (w1, r1, w0); \Uparrow (w1); \Uparrow (w0, r0, w1)\}$
16	dADF-RaW-AC	0(0,2)	$\{\Updownarrow (w0); \Uparrow^{AC} (r0, w1); \Uparrow^{AC} (r1, w0); \Downarrow^{AC} (r0, w1); \Downarrow^{AC} (r1, w0)\}$
17	GalPat	1(0,2)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \Uparrow_{-v} (r0, r1_v), w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \Uparrow_{-v} (r1, r0_v), w1_v)\}$
18	GalRow	1(0,3)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \Uparrow_{R-v} (r0, r1_v), w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \Uparrow_{R-v} (r1, r0_v), w1_v)\}$
19	GalCol	1(0,3)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \Uparrow_{C-v} (r0, r1_v), w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \Uparrow_{C-v} (r1, r0_v), w1_v)\}$
20	Gal9R	0(0,12)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \square(r0, r1_v), w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \square(r1, r0_v), w1_v)\}$
21	Butterfly	1(0,4)	$\{\Updownarrow (w0); \Uparrow (w1_v, \Uparrow_{BF} (r0, r1_v), w0_v); \Updownarrow (w1); \Uparrow (w0_v, \Uparrow_{BF} (r1, r0_v), w1_v)\}$
22	Walk 1/0	1(0,5)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \Uparrow_{-v} (r0), r1_v, w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \Uparrow_{-v} (r1), r0_v, w1_v)\}$
23	WalkRow	1(0,6)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \Uparrow_{R-v} (r0), r1_v, w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \Uparrow_{R-v} (r1), r0_v, w1_v)\}$
24	WalkCol	1(0,6)	$\{\Updownarrow (w0); \Uparrow_v (w1_v, \Uparrow_{C-v} (r0), r1_v, w0_v); \Updownarrow (w1); \Uparrow_v (w0_v, \Uparrow_{C-v} (r1), r0_v, w1_v)\}$
25	HamWh	1(0,7)	$\{\Updownarrow (w0); \Uparrow (r0, w1^h, r1); \Uparrow (r1, w0^h, r0); \Uparrow (r0, w1^h, r1); \Uparrow (r1, w0^h, r0)\}$
26	HamRh	1(0,8)	$\{\Updownarrow (w0); \Uparrow (r0, w1, r1^h, r1); \Uparrow (r1, w0, r0^h, r0); \Uparrow (r0, w1, r1^h, r1); \Uparrow (r1, w0, r0^h, r0)\}$
27	HamWDhrc	1(0,9)	$\{\Updownarrow (w0); \nearrow (w1_v^h, \Uparrow_{R-v} (r0), r1_v, \Uparrow_{C-v} (r0), r1_v, w0_v);$ $\Uparrow (w1); \nearrow (w0_v^h, \Uparrow_{R-v} (r1), r0_v, \Uparrow_{C-v} (r1), r0_v, w1_v)\}$
28	HamWDhc	1(0,10)	$\{\Uparrow (w0); \nearrow (w1_v^h, \Uparrow_{C-v} (r0), w0_v); \Uparrow (w1); \nearrow (w0_v^h, \Uparrow_{C-v} (r1), w1_v);$
29	MOVI	0(0,3)	$\{i_0^{N-1}[\Downarrow^i (w0); \Uparrow^i (r0, w1, r1); \Uparrow^i (r1, w0, r0); \Downarrow^i (r0, w1, r1); \Downarrow^i (r1, w0, r0)]\}$
30	DELAY	0(0,2,14)	$\{\Updownarrow (w0); Del50; \Uparrow (r0, w1); Del50; \Downarrow (r1, w0)\}$

$\Uparrow^{AC}$: Address Complement addressing
$\Uparrow^i$: 2^i addressing
$\Uparrow_v$: all cells except the v-cell
$\Uparrow_{R-v}$: all cells in the same row as the v-cell, except the v-cell
$\Uparrow_{C-v}$: all cells in the same column as the v-cell, except the v-cell
$\Uparrow_{BF}$: cells with a distance of 2^k to the North, East, South and West of the v-cell
N: number of memory address bits
k: Butterfly maximum distance
$Del50$ and $Del100$: 50 and 100 ms delay elements
$\nearrow$: all cells on the main diagonal
h: number of Hammer operations
$\square$: 8 neighbor cells

010, **101**, 011, and **100** [10], [8]; each bold address is the 1's complement of the preceding address.
The 2^i CM is yet another CM; typically used by the MOVI algorithm [10], [8]. It repeats the PMOVI algorithm *N* times (*N* = is 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 0s (i.e., 0000.../0000...) or all 1s.
Checkerboard (bDB): 0101.../1010.../0101.../1010...
Column Stripes (cDB): 0101.../0101.../0101.../0101...
Row Stripes (rDB): 0000.../1111.../0000.../1111...
A 'w0' means that the selected DB is applied; a 'w1' means that the *inverse* of that DB is applied.

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.,

MATS+ (Alg #2): $\{\Updownarrow(w0);\Uparrow(r0,w1);\Downarrow(r1,w0)\}$
March C- (Alg #3): $\{\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);$ $\Downarrow(r0,w1);\ \Downarrow(r1,w0);\Updownarrow(r0)\}$

The first and second MEs of both tests are the same; the third ME of MATS+ 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 $\Updownarrow(rD, w\overline{D})$ where the data value 'D' can be either 0 or 1, and $\Updownarrow$ can be either $\Uparrow$ or $\Downarrow$, except the first and the last MEs, Hence, March C- can be implemented with three *Generic March Elements (GMEs)*: $\Updownarrow(wD)$, $\Updownarrow(rD,w\overline{D})$ and $\Updownarrow(rD)$; MATS+ requires only the first two GMEs. 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 filed in the commands is used to specify the GME. In this

TABLE II. Generic March Elements

B	GME#	GME Description	Alg.#
0	0	$\Updownarrow (wD)$	1-30
0	1	$\Updownarrow (rD)$	1,3-4,10-13
0	2	$\Updownarrow (rD, w\overline{D})$	2-3,9-10,16,30
0	3	$\Updownarrow (rD, w\overline{D}, r\overline{D})$	4-5,8,29
0	4	$\Updownarrow (rD, w\overline{D}, r\overline{D}, wD, rD, w\overline{D})$	6-8
0	5	$\Updownarrow (rD, w\overline{D}, wD)$	6,8
0	6	$\Updownarrow (rD, w\overline{D}, wD, w\overline{D})$	6-8
0	7	$\Updownarrow (rD, w\overline{D}, r\overline{D}, wD)$	7,9-10,14
0	8	$\Updownarrow (rD, w\overline{D}, wD, w\overline{D}, r\overline{D})$	11
0	9	$\Updownarrow (rD, rD, wD, rD, w\overline{D})$	12
0	10	$\Updownarrow (rD, wD, rD, rD, w\overline{D}, r\overline{D})$	13
0	11	$\Updownarrow (wD, rD, w\overline{D})$	15
0	12	$\Updownarrow_v (wD_v, \square(r\overline{D}, rD_v), w\overline{D}_v)$	20
0	13	$\Updownarrow (rD, rD)$	14
0	14	Del50	30
0	15	Del100	8
1	0	$\Updownarrow (wD)$	1-30
1	1	$\Updownarrow (rD)$	1,3-4,10-13
1	2	$\Uparrow_v (wD_v, \Uparrow_{-v} (r\overline{D}, rD_v), w\overline{D}_v)$	17
1	3	$\Uparrow_v (wD_v, \Uparrow_{X-v} (r\overline{D}, rD_v), w\overline{D}_v)$	18-19
1	4	$\Uparrow (wD_v, \Uparrow_{BF} (r\overline{D}, rD_v), w\overline{D}_v)$	21
1	5	$\Uparrow_v (wD_v, \Uparrow_{-v} (r\overline{D}), rD_v, w\overline{D}_v)$	22
1	6	$\Uparrow_v (wD_v, \Uparrow_{X-v} (r\overline{D}), rD_v, w\overline{D}_v)$	23-24
1	7	$\Uparrow (rD, w\overline{D}^h, r\overline{D})$	25
1	8	$\Uparrow (rD, w\overline{D}, r\overline{D}^h, r\overline{D})$	26
1	9	$\nearrow (wD_v^h, \Uparrow_{R-v} (r\overline{D}), rD_v, \Uparrow_{C-v} (r\overline{D}), rD_v, w\overline{D}_v)$	27
1	10	$\nearrow (wD_v^h, \Uparrow_{C-v} (r\overline{D}), w\overline{D}_v)$	28

$x \in \{R, C\}$; Data Value $D \in \{0, 1\}$; $\overline{D}$ = inverse of D

proposal 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 the same in 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 **B**ank, 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, PMOVI (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

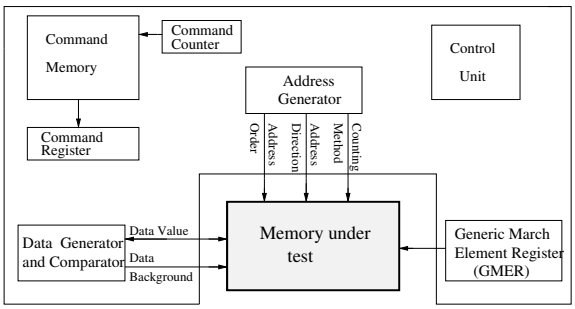

Fig. 2. High-level block diagram

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 1-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 $\Uparrow$ or $\Downarrow$ AO.
5) DVR[0]: *Data Value Register*. Specifies the Data

TABLE III. GME-MBIST engine registers

Reg. Name	BE	Description
CC[5..0]	B	Command Counter
CM[63..0,3..0]	B	Command Memory
CR[7..0]	B	Command Register
AOR[0]	B	Address Order Register
DVR[0]	B	Data Value Register
CMADR[1..0]	B	Counting Method & Addr. Direction Reg.
DBR[1..0]	B	Data Background Register
GMER[3..0]	B	Generic March Element Register
BR[0]	E	Bank Register
REPR[5..0]	E	Re-execute Entry Point Register
RCNTR[3..0]	E	Re-execute CouNT Register
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 selcted. 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 RCNT 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#0 specified in the GMER (which is '$\Updownarrow (wD)$'). 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 a GME 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 $\{_r\Uparrow(w0);\ _r\Uparrow(r0,w1);\ _c\Downarrow(r1,w0,r0)\}$ requires the use of the SAODVE command, because the AD of the GME '$_c\Downarrow(r1,w0,r0)$' 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 REPeat 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 RCNT# 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- $\{\Uparrow(w0);\ \Uparrow(r0,w1);\ \Uparrow(r1,w0);\ \Downarrow(r0,w1);\ \Downarrow(r1,w0);\ \Uparrow(r0)\}$; applied with bDB and Lr. This implementation uses only the *basic* commands.

- $\Uparrow(w0)$. INIT: AO=$\Uparrow$, DV=0, CMADR=Lr, DBR=bDB. Implicit application of GME#0.
- $\Uparrow(r0,w1)$. SGME: AO=$\Uparrow$, DV=0, GMER= 2.
- $\Uparrow(r1,w0)$. SGME: AO=$\Uparrow$, DV=1, GMER= 2.
- $\Downarrow(r0,w1)$. SGME: AO=$\Downarrow$, DV=0, GMER= 2.
- $\Downarrow(r1,w0)$. SGME: AO=$\Downarrow$, DV=1, GMER= 2.
- $\Uparrow(r0)$. SGME: AO=$\Uparrow$, 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 bDB and Lr, while using the *extended* command 'SAODV':

- $\Uparrow(w0)$. INIT: AO=$\Uparrow$, DV=0, CMADR=Lr, DBR=bDB. Implicit application of GME#0.
- $\Uparrow(r0,w1)$. SGME: AO=$\Uparrow$, DV=0, GMER=2.
- $\Uparrow(r1,w0)$. SAODV: AO=$\Uparrow$, DV=1. Note: the GME# of the previous command applies.
- $\Downarrow(r0,w1)$. SAODV: AO=$\Downarrow$, DV=0.
- $\Downarrow(r1,w0)$. SAODV: AO=$\Downarrow$, DV=1.
- $\Uparrow(r0)$. SGME: AO=$\Uparrow$, 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: $\{\Uparrow(w0);\ \Uparrow_v(w1_v,\ \Uparrow_{-v}(r0,r1_v),\ w0_v);\ \Uparrow(w1);\ \Uparrow_v(w0_v,\ \Uparrow_{-v}(r1,r0_v),\ w1_v)\}$; applied with sDB and Lr, as follows:

- LBR: B=1. Note: Bank-1 of the set of GMEs has to be selected.
- $\Uparrow(w0)$. INIT: AO=$\Uparrow$, DV=0, CMADR=Lr, DBR=sDB. Apply implicit GME#0.
- $\Uparrow_v(w1_v,\ \Uparrow_{-v}(r0,r1_v),\ w0_v)$. SGME: AO=$\Uparrow$, DV=1, GMER=2.
- $\Uparrow(w1)$. SGME: AO=$\Uparrow$, DV=1, GMER=0.
- $\Uparrow_v(w0_v,\ \Uparrow_{-v}(r1,r0_v),\ w1_v)$. SGME: AO=$\Uparrow$, DV=0, GMER=2.

Five commands require 10 nibbles of ComMem [13].

Example 4: REPeat(PMOVI & MATS+).
PMOVI: $\{\Downarrow(w0);\ \Uparrow(r0,w1,r1);\ \Uparrow(r1,w0,r0);\ \Downarrow(r0,w1,r1);\ \Downarrow(r1,w0,r0)\}$.
MATS+: $\{\Uparrow(w0);\ \Uparrow(r0,w1);\ \Downarrow(r1,w0)\}$.
This example demonstrates the use of the *REP command* applied to two tests: PMOVI & MATS+. The first time they are executed using **Lr** (**L**inear addressing & Fast-**r**ow), together with sDB (solid DB). Next, they will be re-executed for Lr & bDB, Lr & rDB, Lr & cDB, and for Lc & sDB, Lc & bDB, 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

- $\Downarrow(w0)$. INIT: AO=$\Downarrow$, DV=0, CMADR=Lr, DBR=sDB. Apply implicit GME#0. Note that RCNT will be implicitly cleared here.
- $\Uparrow(r0,w1,r1)$. SGME: AO=$\Uparrow$, DV=0, GMER=3.
- $\Uparrow(r1,w0,r0)$. SAODV: AO=$\Uparrow$, DV=1.
- $\Downarrow(r0,w1,r1)$. SAODV: AO=$\Downarrow$, DV=0.
- $\Downarrow(r1,w0,r0)$. SAODV: AO=$\Downarrow$, DV=1.

MATS+

- $\Uparrow(w0)$. SGME AO=$\Uparrow$, DV=0, GMER=0.
- $\Uparrow(r0,w1)$. SGME: AO=$\Uparrow$, DV=0, GMER=2.
- $\Downarrow(r1,w0)$. SAODV: AO=$\Downarrow$, DV=1.
- REPeat
  REP: CNTR=8, Lc-cDB, Lc-rDB, Lc-bDB, Lc-sDB, Lr-cDB, Lr-rDB, Lr-bDb.

Nine commands require 22 nibbles of command memory for 2 tests executed 8 times [13].

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

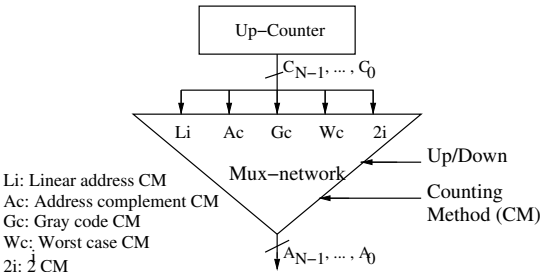

Fig. 3. Up-counter based address generator

TABLE IV. Address Counting Methods(CMs)

Step	Li	Ac	Gc	$2^i = 4$	Pr	Wc
0	0000	0000	0000	0**0**00	0000	-
1	0001	**1111**	0001	0**1**00	0001	000**1**
2	0010	0001	0011	1**0**00	0011	0000
3	0011	**1110**	0010	1**1**00	0111	000**1**
4	0100	0010	0110	0**0**01	1111	-
5	0101	**1101**	0111	0**1**01	1110	00**1**0
6	0110	0011	0101	1**0**01	1101	0000
7	0111	**1100**	0100	1**1**01	1010	00**1**0
8	1000	0100	1100	0**0**10	0101	-
9	1001	**1011**	1101	0**1**10	1011	0**1**00
10	1010	0101	1111	1**0**10	0110	0000
11	1011	**1010**	1110	1**1**10	1100	0**1**00
12	1100	0110	1010	0**0**11	1001	-
13	1101	**1001**	1011	0**1**11	0010	**1**000
14	1110	0111	1001	1**0**11	0100	0000
15	1111	**1000**	1000	1**1**11	1000	**1**000

Note: Li= Linear; Ac= Address Complement; Gc= Gray code; Pr= Pseudo random; Wc= Worst Case Gate Delay

(Ne) AddrGen will be presented. Finally, all the results will be summarized.

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 the '$\Downarrow$' address sequence is generated, by selecting the Q or the $\overline{Q}$ output of bit$_x$ to control the J-K inputs of bit$_{x+1}$. Note that the control of each J-K input requires **two** gates which are in the critical signal path.

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 *even steps* (see column 'Step') of this sequence form a linear $\Uparrow$ address sequence; the addresses for the *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$_0$ controls the muxes of all bits$_x$, with $x > 0$.

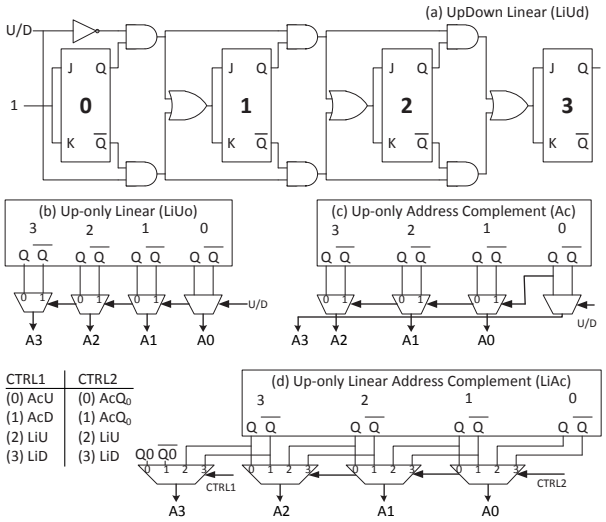

Fig. 4. Linear & Address Compl. AGs

LiAc: Combined LiUo & Ac AddrGen, see Figure 4(d) Figure 4d) shows the implementation of the combined Li&Ac AddrGen. The signals CTLR1 and CTRL2 control the MUXs and are explained in the left part of the figure; e.g., CTRL1=0 means AcUp, etc.

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 address bit size N=8, 12, 16, 20 and 24 bits.

The area increase with increasing N is apparent. The rows '$\triangle$Area Freq in %' list the area increase - in % - when increasing the frequency from 555 to 1111 MHz. Increasing the frequency does not increase the number of gates; however, in order to meet the required clock frequency, certain gates are made larger to get more drive strength; hence, more area overhead. The table clearly shows that the LiUd AddrGen has the largest area increase, which is between 30.7 and 45.3%. Moreover, the table reveals that that LiUd AddrGen requires the largest area;

TABLE V. Area metrics of Li & Ac AGs

Cntr	Freq in MHz	N				
		8	12	16	20	24
LiUd	555	123	186	262	344	426
LiUd	833	135	219	305	401	500
LiUd	1111	179	265	360	455	556
△Area Freq in %		45.3	41.9	37.2	32.3	30.7
LiUo	555	107	170	230	286	352
LiUo	833	110	172	234	297	365
LiUo	1111	116	191	274	355	435
△Area Freq in %		8.4	12.6	19.4	24.0	23.6
△Area LiUd-Uo in %		35.2	27.9	23.8	22.0	21.8
Ac	555	108	168	227	289	351
Ac	833	112	171	230	299	362
Ac	1111	114	192	273	353	435
△Area Freq in %		5.3	13.8	20.2	22.3	24.1
LiAc	555	122	182	252	325	388
LiAc	833	134	202	269	341	414
LiAc	1111	139	227	313	396	486
△Area Freq in %		14.1	24.8	24.3	22.0	25.1

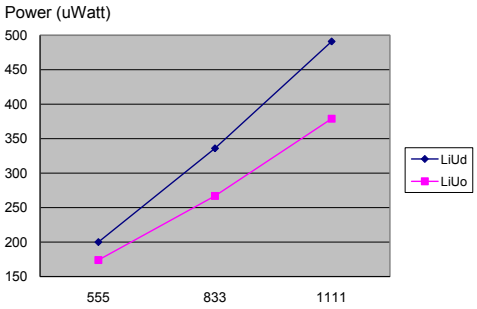

Fig. 5. Power in uW for LiUd & LiUo

e.g., depending on the frequency, LiUd consumes 21.8 to 35.2% more than LiUo AddrGen; this is given in row '△Area LiUd-Uo in %'.

Figure 5 shows the power requirements for the LiUd and the LiUo AddrGens; the LiUd is worse, especially for higher frequencies, by 13 to 23%. The power increases non-linearly with the frequency, because a higher frequency also demands a larger circuit area; see Table V. Considering the advantages the LiUo counter has over the LiUd counter, the latter will not be considered any more from this point on.

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 2^i (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: bit_0 of the Gc address can be derived from bit_0 of the Linear address by inverting it when bit_1 of the linear address is '1'. This is shown in Figure 6(a): the mux of bit_0 is controlled by the signal 'Q1'. Similar reasoning applies to bit_1 and bit_2. The mux of bit_3 is controlled by the Up/Down signal,

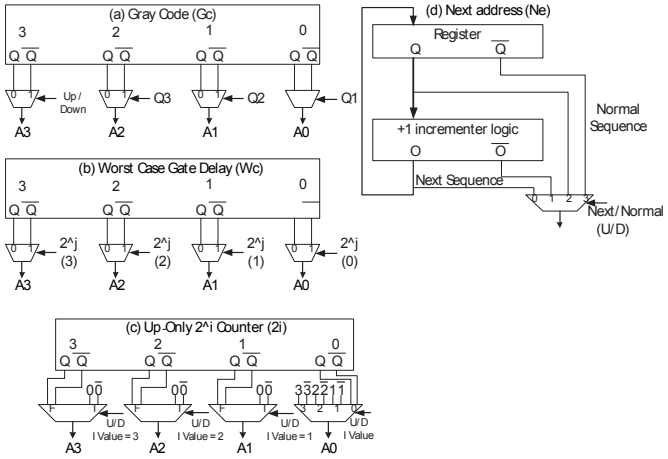

Fig. 6. Gc, Wc, 2i and Ne AddrGens

which means that in case of the ⇑ address sequence, 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 2^N victim addresses ($v_{addr}es$), the following N address-triplets are generated: $v_{addr} \oplus 2^j, v_{addr}, v_{addr} \oplus 2^j$; for $0 \leq j \leq N-1$. 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 $\overline{Qj}$ output, under control of the corresponding mux with control input '2^ j'. For example, for bit_2 the mux control input is labeled '2^ 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 2^i AddrGen
The column '2^i=4' of Table IV shows the 2^i 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 muxes, each with N inputs, to transform the Li address sequences into the 2^i sequences. However, this requires a total of $N*N=N^2$ mux-inputs. An economical solution is shown in Figure 6(c); it has one mux for bit_0 with N inputs, and muxes with 2 inputs for all other bits. For all values of i, except for $i = 0$, the muxes interchange col_i with col_0. Therefore, the mux for bit_0 requires N inputs, while the other muxes only require 2 inputs.

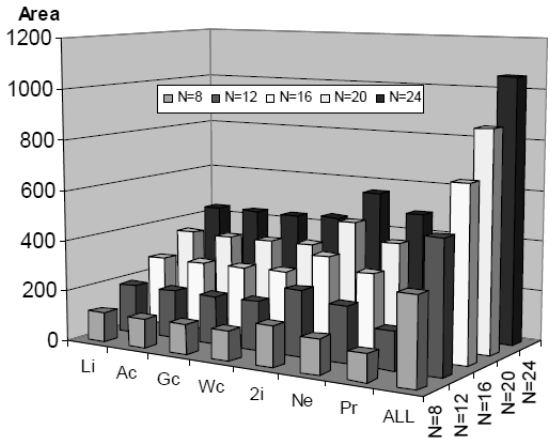

Fig. 7. Area for different AddrGens

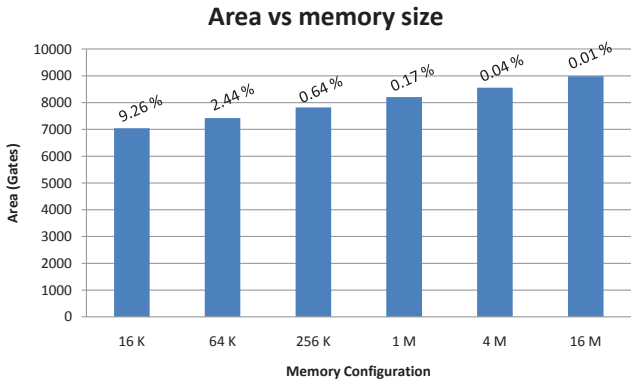

Fig. 8. Area overhead versus memory size

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 completness, 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 ($4\mu m^2$ 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 overhaed becomes negligeable 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 – stresse 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 2^i 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

TABLE VI. Comparison with other MBISTs

	Appello 03 [25]	Du 05-06 [3], [6]	Park 09 [16]	GME MBIST
March	✓	✓	✓	✓
Galloping/	x	✓	✓	✓
Walking	x	✓	✓	✓
Butterfly	x	✓	✓	✓
Hammer	x	x	x	✓
Sliding Diag.	x	✓	✓	x
AD Open	x	✓	x	✓
AD Delay	x	x	x	✓
Byte WR Enb.	x	✓	x	x
Moving Inv.	x	x	x	✓
Retention	✓	x	x	✓
Multi-level nested loop	x	✓(4)	✓(2)	✓(2)
Data Background	s,cb DB Reg	s,cb,r/c Addr. Unq.	NA	s,cb,r/c
Counting Method	U/D fixed	Lr, Lc, Ld	NA	Lr, Lc, Ld AC, 2^i
Command Memory	43x4-b 8x8-b		8x9-b	64x4-b, 16x4-b
Total bits for March C+	160	216	126	36
Technology	0.18 μm	0.13 μm 90 nm [3]	0.13 μm	90 nm
Freq (MHz)	40	NA, 333 [3]	300	500
AREA for 16Kx16-b mem.	7.9 K gates			5.6 Kgates
AREA for 8Kx32-b mem.		13.6 Kgates [6]	6.4 Kgates No diagnosis	3.4 Kgates

(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 flexiblity such as supporting address complement and 2^i counting methods.

References

[1] R.C. Aitken, 'A Modular Wrapper Enabling High Speed BIST and Repair for Small Wide Memories', *In Proc. of the IEEE Int. Test Conf.*, paper 35.2, pp. 997- 1005, 2004.

[2] T.J. Powell, et al., 'BIST for Deep Submicron ASIC Memories with High Performance Application', *In Proc. of the IEEE Int. Test Conf.*, pp. 386-392, 2003.

[3] X. Du, N. Mukherjee and T.M.Cheng, 'Full-Speed Field-Programmable Memory BIST Architecture', *In Proc. of the IEEE Int. Test Conf.*, paper 45.3, 2005.

[4] Z. Conroy, et.al, 'A practical perspective on reducing ASIC NTFs', *Proc. of Int. Test Conference*, pp. 349, 2005.

[5] L. Dilillo, et.al, 'Dynamic read destructive fault in embedded-SRAMs: analysis and march test solution', *Proceedings. Ninth IEEE European Test Symposium*, pp. 140-145, 2004.

[6] X. Du, N. Mukherjee, W-T Cheng, S. M. Reddy 'A Field-ProgrammableMemory BIST Architecture Supporting Algorithms and Multiple Nested Loops', *Proc. of the Asian Test Symposium*, paper 45.3, 2006.

[7] J.B. Khare, A.B. Shah, A. Raman and G. Rayas, 'Embedded Memory Field Returns - Trials and Tribulations', *Proc. of Int. Test Conference*, pp. 1-6, 2006.

[8] M. Klaus and A. J. van de Goor, 'Tests for Resistive and Capacitive Defects in Address Decoders', *Proc. of the 10th Asian Test Symposium*, pp. 31-36, 2001.

[9] T. Powell, et.al, 'Chasing subtle embedded RAM defects for nanometer technologies', *Proc. of Int. Test Conference*, pp. 860, 2005.

[10] A.J. van de Goor, Testing Semiconductor Memories: Theory and Practice, *ComTex Publishing, The Netherlands*, 1998, Ad.vd.Goor@kpnplanet.nl.

[11] A.J. van de Goor, S. Hamdioui and G. N. Gaydadjiev, 'New Algorithms for Address Decoder Delay Faults and Bit Line Imbalance faults', *Proc. of the 18th Asian Test Symposium*, pp. 31-36, 2009.

[12] S. Hamdioui, Z. Al-Ars, A.J. van de Goor, M. Rodgers, 'Dynamic Faults in Random-Access-Memories: Concept, Fault Models and Tests', *Journal of Electronic Testing: Theory and Applications*, pp. 195-205, April 2003.

[13] H. Kukner, 'Generic and Orthogonal March Element based Memory BIST Engine' Master Thesis, CE-MS-2010-01, Delft University of Technology, September 2010.

[14] Faraday Corp., Faraday Technology Corp., FSD0A A SH 90 nm Synchronous High Density Single-port SRAM Compiler, Oct. 2006.

[15] Synopsys Corp., Synopsys Inc., Design Compiler 2010, v. D-2010.03, February, 2010.

[16] Y. Park, J. Park, T. Han, and S. Kang, An Effective Programmable Memory BIST for Embedded Memory, IEICE Transactions on Information and Systems E92-D (2009), no. 12, 2508-2511.

[17] S. Hamdioui, A.J. van de Goor, J.D. Reyes, and M.Rodgers, 'Memory test experiment: industrial results and data', *IEE Proc. of Computers and Digital Techniques*, Vol. 153 , Issue: 1, pp. 1-8, 2006.

[18] A.J. van de Goor, S. Hamdioui, G. Gaydadjiev and Z. Alars 'Generic March Element Based Memory Built-In Self Test', Dutch Patent Application; Filing Number NL 2004407, Filed date: Jan 2010.

[19] R. Nair, 'An Optimal Algorithm for Testing Stuck-at Faults Random Access Memories', IEEE transactions on Computers, Vol. C-28, No. 3, pp. 258-261, 1979.

[20] A. J. van de Goor, S. Hamdioui and R. Wadsworth, 'Detecting Faults in the Peripheral Circuits and an Evaluation of SRAM Tests', *In Proc. of the IEEE Int. Test Conf*, pp. 114-123, 2004.

[21] A.J. van de Goor and A. Paalvast, 'Industrial Evaluation of DRAM SIMM Tests', *In Proc. IEEE Int. Test Conf.*, pp. 426-435, 2000.

[22] S. Hamdioui, Z. Al-Ars and A.J. van de Goor, 'Opens and Delay Faults in CMOS RAM Address Decoder', *IEEE Trans. on Computers*, pp. 1630-1639, December 2006.

[23] L. Dilillo, et.al, 'ADOFs and Resistive-ADOFs in SRAM Address Decoders: Test Conditions and March Solutions', *Jour of Electronic Testing: Theory and Applications*, Vol. 22(3), pp. 287-296, June 2006.

[24] J.H. De Jonge and A.J. Smeulders, "Moving Inversions Test Pattern is Thorough, Yet Speedy", In Comp. Design, pp. 169-173, 1976.

[25] D. Appello, et al., 'Exploiting Programmable BIST For The Diagnosis of Embedded Memory Cores', *Int. Test Conference*, pp. 379, 2003.