Optimizing Memory BIST Address Generator Implementations

Ad J. van de Goor[1,2]

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

Halil Kukner[2] Said Hamdioui[2]

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

***Abstract*— Memory Built-In Self-Test (MBIST) has become a standard industrial practice. Its quality is mainly determined by its fault detection capability in relationship to the the area overhead. The MBIST Address Generator (AG) is largely responsible for the fault detection capability, and has a significant contribution to the area overhead. This paper analyzes the properties and implementation aspects of several AGs. In addition, it presents a novel, very systematic, high-speed, low-power and low-overhead implementation, based on an Up-counter and a set of multiplexors.**

Keywords: *Memory BIST, Address Generator, up-only counter, area, power, implementation aspects*

I. Introduction

Memory Built-In Self-Test (MBIST) has become a standard industrial practice [1], [4], [5], [15], [12]. MBIST is important because memory cores are a major part of the die area; it is forecasted that by 2014 they will occupy 94% of the die area [16]. In addition, they are designed with minimal design rules, making them more susceptible to defects, and hence, to faults. For a high product quality, the fault detection capability of the MBIST is critical.

In the world of MBIST, memory accesses have to be applied *at-speed*, using *Back-to-Back* (BtB) memory cycles [2]-[5], [7]-[9]. Systems require *large, high speed* memories, while current technology exhibits a large spread in implementation parameters, resulting in speed-related (i.e., delay) faults [3], [8], [11], [19]. Their detection is mandatory in today's industry [2], [4], [6], [20], and requires *non-linear algorithms* such as GalPat, GalRow and GalColumn, and a special **A**ddress **G**enerator (**AG**). The AG is a key MBIST component. In order to detect speed-related faults, the AG has to generate a large set of address sequences, with BtB cycles and the appropriate address transitions. Its complexity is a major design issue, since it requires a large area and limits the MBIST speed.

Figure 1 shows the relative area -in % - taken by the five components for three MBIST designs [4], [15], [12]: Control (Ctrl), test algorithm Memory (Memory), Instruction fetch and decode (Instr), Address Generator (AddrGen), and Data Generator (DataGen). Although the designs are very different, the area requirement of the AG is significant: between 26 and 33%. Reducing the algorithm Memory, which takes between 38 and 42% of the MBIST area, has been addressed in [21]. A brute-force implementation can be very costly.

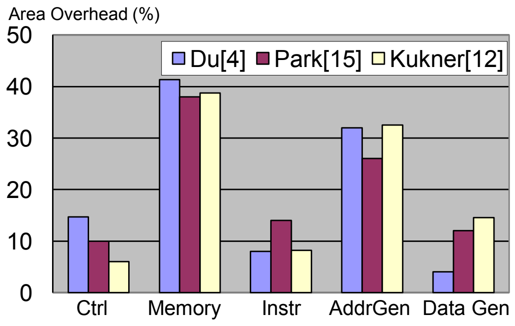


Fig. 1. AG-Overhead

In [17] the authors reported that an MBIST redesign, using innovative ideas, can result in an area reduction of 75%; the AG was a major contributor to this area saving.
This paper contributes to the area of MBIST implementation by emphasizing its most critical part: the **A**ddress **G**enerator (**AG**). Therefore, it is of value to the practicing engineer. The main contribution consists of an implementation analysis of AGs to support a variety of address sequences, such as Linear, Address Complement, Gray Code, etc. The most common and important address sequences are supported with a single Up-only counter, together with a set of multiplexers. This results in significant savings in area and power, and allows for a higher speed MBIST engine and a very systematic implementation. E.g., a 24-bit Linear AG, implemented with an Up-only counter and a set of muxes, shows 21.8% area and 23% power savings, as compared with an implementation, using an Up-Down counter.

The organization of this paper is as follows: Section II covers the requirements for AGs; Section III shows the implementation alternatives and analysis for the Linear and Address complement AGs; Section IV covers the Gray code, the Worst Case Gate Delay and the 2^i AGs; Section V discusses the Next address and the Pseudo-random AGs; while Section VI ends with the conclusions.

II. Address generator requirements

There are $N!$ (N-factorial) *Counting Methods* (*CM*s); i.e., ways of counting to N. E.g., for N=3 there are 6 CMs: 012, 021, 102, 120, 210, and 201. For memory testing, the AG has to generate several CMs, since each CM has its own fault detection capability [3], [6], [8], [9], [11], [20]. For this paper, the most common, and important, CMs will be considered; they are explained next. Table I highlights the

TABLE I
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	000**0**
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	00**0**0
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	0**0**00
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	**0**000
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

CMs by giving an example of each CM for N=4 (N is the # of address bits).

- *Linear (Li) CM* specifies the address sequence: 0, 1, 2 , 3, ..., 2^N-1 when going Up '$\Uparrow$'; and 2^N-1,..., 3, 2, 1 , 0 when going Down '$\Downarrow$'. The Li CM is used for detecting single-cell and coupling faults.
- *Address complement (Ac) CM* specifies the address sequence: 0000, **1111**, 0001, **1110**, 0010, **1101**, etc. [10]. The *even steps* in Table I, see column 'Step', of this sequence form a linear $\Uparrow$ address sequence; the addresses of the *odd steps*, in **bold** font, are formed by taking the one's complement of the preceding even steps. The Ac CM stresses the address decoders, because all N or N-1 address bits switch upon an address transition; this causes lots of noise, a large power surge, and maximal delay. It is used for detecting speed-related faults.
- *Gray code (Gc) CM* has address transitions which differ only in *one* bit (i.e., they have a *Hamming distance* of 1); see column 'Gc' in Table I. Its properties are opposite to those of the Ac CM; it causes minimal noise, power and delay, and is used for minimal stress.
- *Worst Case Gate Delay (Wc) CM* derives, for *every* address, N address-triplets, with a Hamming distance of 1, by successively inverting a single address bit. The column 'Wc' in Table I shows the address-triplets only for address '**0000**' [11]. For every address bit, address triplets consisting of (a) the address with the inverted bit, (b) the original address, and (c) the address with the inverted bit, are generated. The Wc CM is used to detect speed-related faults [11].
- 2^i *CM* generates all address pairs with a $Hamming$ distance of 1; i.e., address-pairs which differ in *one* bit. The column $2^i = 4$ in Table I shows the address sequence for $i = 2$; i.e., with address increments/decrements with a value of 4. Note that end-around carry is used when the number under-/over-flows. The 2^i CM is used by the popular MOVing Inversions (MOVI) test [8], [10] for speed-related faults.

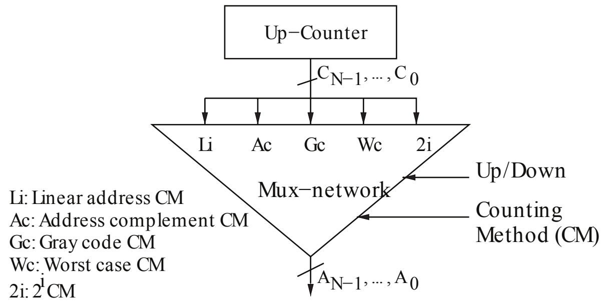


Fig. 2. Up-counter & Mux-network AG

- *Pseudo-random (Pr)* generates a Pr address sequence. It can be used, e.g., to verify the fault coverage of the deterministic tests. The column Pr in Table I lists the address sequence for a 4-bit generator with a characteristic polynomial function: $x^4 + x^1 + 1$ [10].

It will be shown that the Li, Ac, Gc, Wc and the 2^i CMs can be implemented with a single Up-counter (with outputs $C_{N-1}, ..., C_0$), and a Mux-network with Address outputs $A_{N-1}, ..., A_0$; see Figure 2 and Table II. It has control inputs 'U/D' (Up/Down) and the desired CM (Li, Ac, Gc, Wc or 2^i). Note: In Figure 3, 4, 5, 6 and 8, the addresses A_3, A_2, A_1 and A_0 are labeled: Q_3, Q_2, Q_1 and Q_0.

III. LINEAR & ADDRESS COMPLEMENT AGS

This section presents the implementation and the analyses (in terms of area overhead, speed and power consumption) of four AGs: two versions of Li AGs, the Ac AG and a combined version of the Li and the Ac AG.

A. Li and Ac AG implementations

The four AGs are shown in Figure 3 and are explained next. Note that all examples use a 4-bit implementation, which is sufficient to show the concept, while preserving space.

LiUd: Linear AG based on Up-down counter

Figure 3(a) shows the LiUd AG 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 AG based on Up-only counter

Figure 3(b) depicts the LiUo AG 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.

The left column of Table II lists the CM; the next column the AO $\{\Uparrow$ or $\Downarrow\}$, followed by the four Address bits: A_3, A_2, A_1, A_0. The rows 'Li' describe the equations, implemented via the Mux data and the Mux control inputs. E.g., for the $\Uparrow$ and the $\Downarrow$ AOs: for Li $\Uparrow$, $A_3 = C_3$, while for Li $\Downarrow$, $A_3 = \overline{C_3}$.

TABLE II

MATRIX FOR LI, AC, GC 2I WC & NE AGS

CM	⇑⇓	A_3	A_2	A_1	A_0
Li	⇑	C_3	C_2	C_1	C_0
Li	⇓	$\overline{C_3}$	$\overline{C_2}$	$\overline{C_1}$	$\overline{C_0}$
Ac	⇑	C_0	$C_3 \oplus C_0$	$C_2 \oplus C_0$	$C_1 \oplus C_0$
Ac	⇓	$\overline{C_0}$	$C_3 \oplus C_0$	$C_2 \oplus C_0$	$C_1 \oplus C_0$
Gc	⇑	C_3	$C_2 \oplus C_3$	$C_1 \oplus C_2$	$C_0 \oplus C_1$
Gc	⇓	$\overline{C_3}$	$C_2 \oplus C_3$	$C_1 \oplus C_2$	$C_0 \oplus C_1$
Wc	⇑	$C_3 \oplus (j=3)$	$C_2 \oplus (j=2)$	$C_1 \oplus (j=1)$	$C_0 \oplus (j=0)$
Wc	⇓	$C_3 \oplus (j=3)$	$C_2 \oplus (j=2)$	$C_1 \oplus (j=1)$	$C_0 \oplus (j=0)$
2i;0	⇑	C_3	C_2	C_1	C_0
2i;0	⇓	$\overline{C_3}$	$\overline{C_2}$	$\overline{C_1}$	$\overline{C_0}$
2i;1	⇑	C_3	C_2	C_0	C_1
2i;1	⇓	$\overline{C_3}$	$\overline{C_2}$	$\overline{C_0}$	$\overline{C_1}$
2i;2	⇑	C_3	C_0	C_1	C_2
2i;2	⇓	$\overline{C_3}$	$\overline{C_0}$	$\overline{C_1}$	$\overline{C_2}$
2i;3	⇑	C_0	C_2	C_1	C_3
2i;3	⇓	$\overline{C_0}$	$\overline{C_2}$	$\overline{C_1}$	$\overline{C_3}$

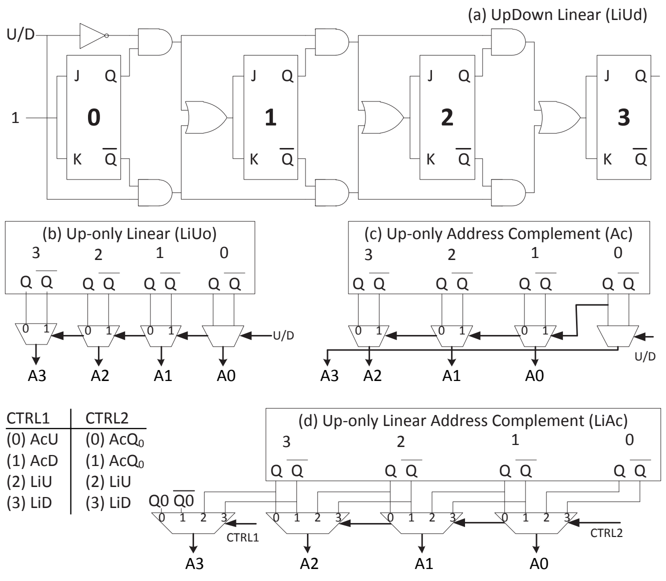


Fig. 3. Linear & Address Compl. AGs

Ac: Address complement AG

Column 'Ac' of Table I shows a 4-bit address sequence for the Ac CM. Figure 3(c) shows Ac AG implementation using an Up-only counter. The 'U/D' control signal controls the most-significant address bit 'A_3', which is the least-significant counter bit 'C_0', because A_3 of the Ac CM changes with every clock period; see Table I. The Q output of C_0 controls the muxes of all A_x, with $0 \leq x < 3$.

Rows 'Ac' of Table II describe the Mux functionality; e.g., $A_2 = C_3 \oplus C_0$ is implemented via Mux data input C_3 and control input C_0, see Figure 3(c).

LiAc: Combined LiUo & Ac AG, see Figure 3(d)

This AG uses the control signal 'CTRL1' for the mux of A_3, and 'CTRL2' for the other address bits. E.g., CTRL1=0 means **AcUp**. Similar to Figure 3(c), the Q_0 and $\overline{Q_0}$ data inputs to the left-most mux of Figure 3(d) are used to generate A_3. CTRL1=3 means **LiDown**; similar to Figure 3(b), $\overline{Q_3}$ is connected to the input '3' of the left-most mux in Figure 3(d) to generate A_3.

The CTRL2 inputs are Ac, Q_0 and U/D. For the generation of the Ac CM, the mux inputs '0' and '1' are used. Similar to Figure 3(c), Q_0 controls the generation of $Ac \Uparrow$ sequence via mux input '0' when $Q_0 = 0$, and $Ac \Downarrow$ sequence via mux input '1' when $Q_0 = 1$. , The mux inputs '2' and '3' are used for the generation of the Li CM; the U/D (Up/Down) control signal determines whether mux input '2' or '3' is selected.

TABLE III

AREA METRICS OF LI & AC AGS

AG	Freq in MHz	N (# of address bits) 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
△LiAc-LiUo Area in %		19.8	18.8	14.2	11.6	11.7

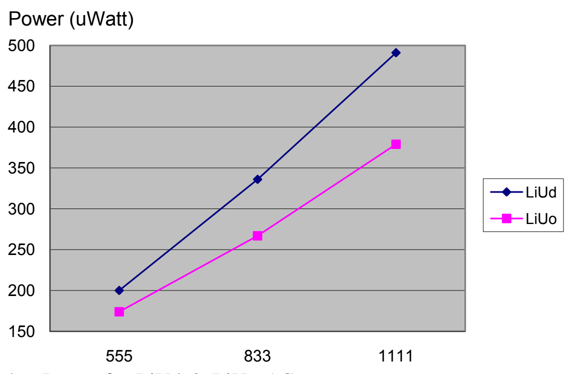


Fig. 4. Power for LiUd & LiUo AGs

B. *Li and Ac AG simulation results*

The AGs are synthesized with the Synopsys Design Compiler [14], using the Faraday UMC 90 nm Standard Process library [13]. Table III shows the area, *in terms of standard 2-input NAND gates*, for the 4 AGs (the LiUd, the LiUo, the Ac, and the LiAc AG). The column 'Freq' lists the three operating frequencies in MHz; the columns thereafter list the area requirements for AGs consisting of 8 ($N = 8$), 12, 16, 20 and 24 address-bits.

Note that the area increase with increasing N (the # of address bits) is apparent. The LiUd AG has the largest area increase: between 30.7 and 45.3% (see table entry "△Area Freq in %"); LiUo has an increase of only 8.4 to 23.6%. Moreover, the table reveals that that LiUd AG consumes the largest area; e.g., depending on the operating frequency, LiUd consumes 21.8 to 35.2% more than the LiUo AG; see row '△Area LiUd-Uo in %'.

The rows '△Area Freq in %' list the percentage of area increase when increasing the frequency from 555 to 1111

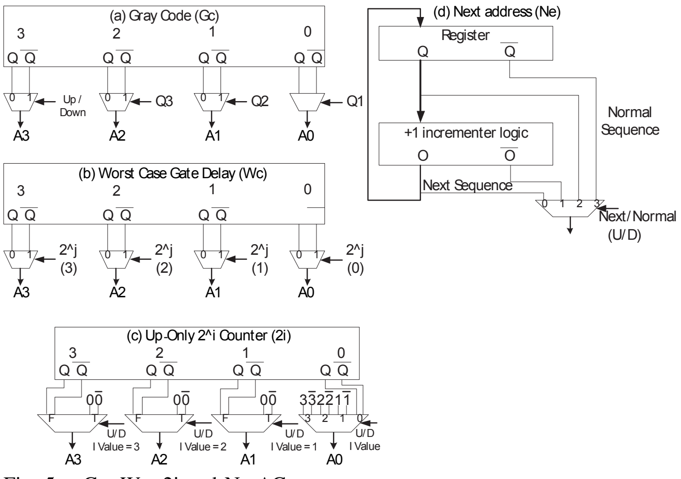


Fig. 5. Gc, Wc, 2i and Ne AGs

MHz. Increasing the frequency does not increase the number of gates required to implement the AG. However, in order to meet the required clock frequency, certain gates are made *larger* to get more drive strength; hence, more area overhead when expressed in terms of # of *standard* 2-input NAND gates.

Figure 4 shows the power requirements for the LiUd and the LiUo AGs; the LiUd is worse, especially for higher frequencies, by 13 to 23%. The power increases non-linearly with the frequency, because higher frequencies also demand a larger circuit area; see Table III. Considering the advantages the LiUo counter has over the LiUd counter, the latter will not be considered any more from this point on.

Increasing the AG capability from Li to include Ac does not double the AG area. Figure 3(d) shows that to each of the N address muxes, 2 extra inputs are added, together with the control of the extra mux inputs. The rows with labels 'LiAc' of Table III show the area requirement for this AG. The row '$\triangle$LiAc-LiUo Area in %' shows the LiUo AG area increase, for Freq = 1111MHz, to implement the Ac capability: this is between 11.6 and 19.8%. This means that adding another CM only marginally increases the AG area.

IV. Gray code, Worst Case Gate Del. & 2^i AGs

This section analyzes the **G**ray **c**ode (Gc), the Worst Case Gate Delay (Wc), and the 2^i AGs; see also Table I.

Gc: Gray code AG; see Figure 5(a)
The column 'Gc' of Table I shows a 4-bit Address Sequence (AS) for the Gc CM. By comparing this sequence with that of Li AG, one can see that the Gc AS can be derived from the Li AS as follows: $A_0 = C_0 \oplus C_1$; i.e., A_0 of the Gc address can be derived from C_0 of the Linear address by inverting it when C_1 of the linear Up-counter is '1'; see also Table II. This is implemented in Figure 5(a) by controlling the mux of A_0 with the signal 'C_1'. A similar reasoning applies to A_1 and A_2. The mux of A_3 is controlled by the Up/Down signal, which means that in case of the $\Uparrow$ address sequence, the '0' input of the mux will select C_3 to generate A_3; see Table I.

TABLE IV
Ways of 2^i addressing

	Regular 2^i CM				Minimal 2^i CM			
#	0	1	2	3	0	1	2	3
0	000**0**	00**0**0	0**0**00	**0**000	0000	00**00**	0**0**0**0**	**0**00**0**
1	000**1**	00**1**0	0**1**00	**1**000	0001	00**10**	0**1**0**0**	**1**00**0**
2	001**0**	01**0**0	1**0**00	**0**001	0010	00**01**	0**0**1**0**	**0**01**0**
3	001**1**	01**1**0	1**1**00	**1**001	0011	00**11**	0**1**1**0**	**1**01**0**
4	010**0**	10**0**0	0**0**01	**0**010	0100	01**00**	0**0**0**1**	**0**10**0**
5	010**1**	10**1**0	0**1**01	**1**010	0101	01**10**	0**1**0**1**	**1**10**0**
6	011**0**	11**0**0	1**0**01	**0**011	0110	01**01**	0**0**1**1**	**0**11**0**
7	011**1**	11**1**0	1**1**01	**1**011	0111	01**11**	0**1**1**1**	**1**11**0**
8	100**0**	00**0**1	0**0**10	**0**100	1000	10**00**	1**0**0**0**	**0**00**1**
9	100**1**	00**1**1	0**1**10	**1**100	1001	10**10**	1**1**0**0**	**1**00**1**
10	101**0**	01**0**1	1**0**10	**0**101	1010	10**01**	1**0**1**0**	**0**01**1**
11	101**1**	01**1**1	1**1**10	**1**101	1011	10**11**	1**1**1**0**	**1**01**1**
12	110**0**	10**0**1	0**0**11	**0**110	1100	11**00**	1**0**0**1**	**0**10**1**
13	110**1**	10**1**1	0**1**11	**1**110	1101	11**10**	1**1**0**1**	**1**10**1**
14	111**0**	11**0**1	1**0**11	**0**111	1110	11**01**	1**0**1**1**	**0**11**1**
15	111**1**	11**1**1	1**1**11	**1**111	1111	11**11**	1**1**1**1**	**1**11**1**

Wc: Worst Case Gate Delay AG; see Figure 5(b)
The column 'Wc' of Table I sketches part of a 4-bit Wc address sequence. The Wc CM requires that for every address, a single address bit has to be inverted; see also Section II. This is accomplished by selecting the C_j or the $\overline{C_j}$ output, under control of the corresponding mux with control input '$j = i$'; see Table 2. For example, for A_2 the mux control input is 'j=2'; indicated in Figuure 5(c) by the mux-control input "$2^j(2)$". Note that of the 4 mux control inputs *only one* is active, such that only *one address bit* is inverted.

2i: 2^i **AG**; see Figure 5(c)
The column '2^i=4' of Table I shows the 2^i address sequence with *address increments/decrements* of 4; i.e., i=2. This CM is important for the MOVI algorithm [8], [10], which is used throughout the industry. It therefore is worth to have an optimal implementation. Table IV will be used to explain the 2^i sequences. The sub-table 'Regular 2^i CM' lists the 'Regular' 2^i CM. Column '0' stands for 'i=0'; hence, address increments/decrements of $2^0 = 1$ are used (see last digit, in **bold** font, of column '0'). In the next column, '1', address increments/decrements of 2^1= 2 are used, etc. A barrel shifter with N muxes, each with N inputs, could be used to transform the Li address sequences into the 'Regular' 2^i sequences. However, this requires a total of $2{*}N{*}N{=}2N^2$ mux inputs for the $\Uparrow$ and the $\Downarrow$ AOs.

A Minimal solution is shown in Figure 5(c); the mux for A_0 has 2*N inputs, and the muxes for A_1, A_2 and A_3 each have 2*2 inputs. This reduces the required number of mux inputs from $2N^2$ to 2*(2*(N-1)+N)=6N-4. The second sub-table of Table IV, 'Minimal 2^i CM', shows the operation. The sequence in the column '0' is identical to the Regular sequence. For all other values of i the muxes interchange col$_i$ with col$_0$; see **bold** digits in the columns. Therefore, the mux for A_0 requires 2*N inputs, while the other muxes only require 2*2 inputs.

Table 2 has N pairs of entries for the 2^i CM; for N=i, A_i=C_0 or $\overline{C_0}$, while A_0=C_i or $\overline{C_i}$. Note that in each column A_i, for $i > 0$, N-1 entry-pairs are identical, requiring only $2 * 2$ mux-inputs.

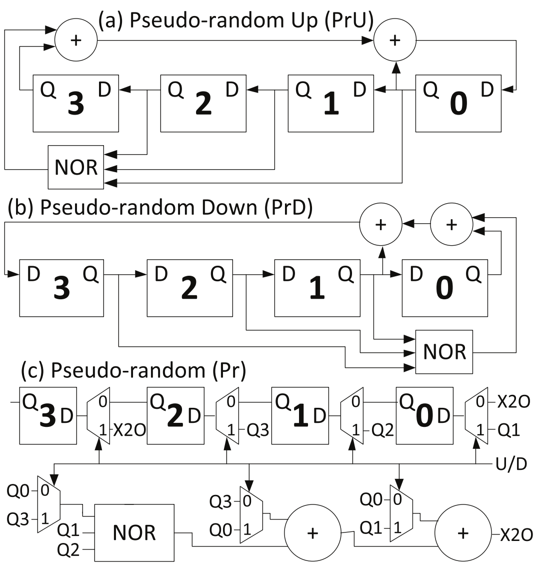


Fig. 6. Pr AGs

V. Next address & Pseudo-random AGs

This section describes the implementation of the **Ne**xt address (Ne) and the **P**seudo-**r**andom (**Pr**) AGs; see Table I, Figure 5(d) and Figure 6. The implementation cannot be done with inputs for the Mux-network of Figure 2.

Ne: Next address AG; see Figure 5(d)
Some algorithms, like those targeting Bit Line Imbalance Faults [18], [11], require the generation of the next address. This means that, within a march element, operations are applied to a given address, as well as to the next address. The Ne AG implementation is based on the idea that the Up-only counter can be split into two units: the 'Register' and the '+1 increment logic', as shown in Figure 5(d). To generate the $\Uparrow$ and $\Downarrow$ sequences, the mux in the figure can select the Register outputs, which represent the 'Normal Sequence', via mux inputs '2' and '3'. Alternatively, the generation of the 'Next Sequence' in the $\Uparrow$ or the $\Downarrow$ direction is done via mux inputs '0' and '1'.

Pr: Pseudo-random AG; see Figure 6(a, b and c)
The implementation of the Pr CM requires a Linear Feedback Shift Register (LFSR), instead of extra inputs to the Mux-network of Figure 2. Figure 6(a) can generate the Address Sequence (AS) of the column 'Pr' of Table I, which we will denote as the Pseudo-random Up 'PrU' AS. For this, the LFSR uses the primitive polynomial 'G(x)' defined as $G(x) = x^4 + x + 1$, such that the maximum-length sequence can be generated [10]. This polynomial is implemented by XORing bit$_3$ and bit$_0$, and feeding it to the input of the LFSR, as shown in Figure 6(b). The LFSR has to shift to the **left**; i.e., towards the most significant address bit. The NOR gate allows for the generation of the all-0 address; when the state of the LFSR is 1000 or 0111, it inserts a '1' into the XOR network. That way it can exit state '0000'.

For the generation of the Pseudo-random Down (PrD) AS, which has to be the exact inverse of the PrU AS, the LFSR has to shift towards the least-significant bit (i.e., to the **right**, while the XOR network has to implement the reverse polynomial $G^*(x)$ which satisfies the equation: $G^*(x) = x^g * G(1/x)$; g is the *degree* of the polynomial [10]. The reverse polynomial $G^*(x) = x^4 * (1/x^4 + 1/x + 1) = x^4 + x^3 + 1$ and its implementation is given in Figure 6(b).

Figure 6(c) shows the 4-bit Pr AG which can generate both the $\Uparrow$ and the $\Downarrow$ sequences; it is a combination of Figure 6(a) and 6(b). The left and right shift capability is supported by the muxes controlled by the Up/Down signal, and located between the LFSR cells.

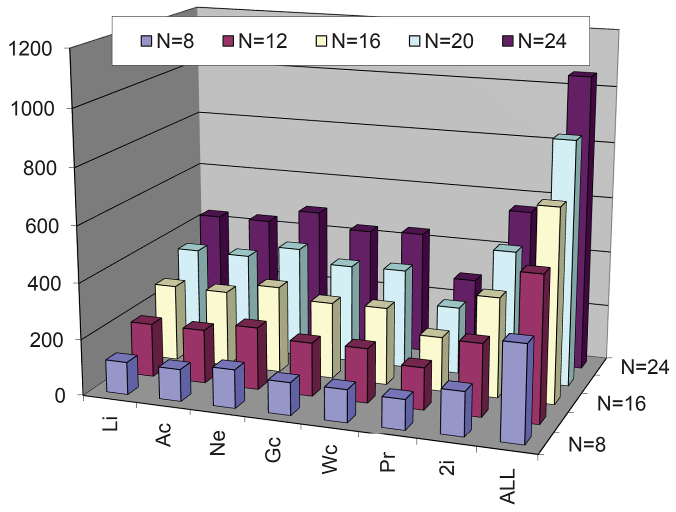


Fig. 7. AG implementation area

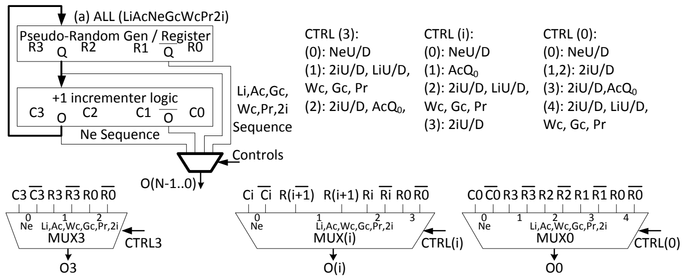
VI. Conclusions and recommendations

This paper analyzes Address Generator (AG) implementation alternatives for Memory BIST. This has been motivated by the fact that the AG takes about 30% of the MBIST engine area. The set of Counting Methods (CMs), commonly used in industry to detect different faults classes, which include speed-related faults, consist of the Linear (Li), Address complement (Ac), Gray code (Gc), Worst Case Gate Delay (Wc), 2^i (2i), Next address (Ne), and the Pseudo-random (Pr) CMs.

The AGs have been designed and implemented in Faraday 90 nm technology. The results show that the Up-Down counter, as compared with an Up-only counter with multiplexers, is less area efficient (by 22 to 35%) and also less power efficient (by 13 to 23%). Furthermore, it has been shown that the optimal AG implementation is based on the use of an **Up-counter**, with a set of **multiplexers**. This implementation can easily be extended to support additional CMs, which make the design and implementation of the AG more systematic, and less area and power demanding.

The Next address CM is supported very economically by splitting the Up-only counter into a 'Register' and a '+1 increment logic unit', while the Pseudo-random CM is supported by modifying the 'Register' to become a Linear Feedback Shift Register.

Figure 7 depicts the area overhead required for each of the seven CMs covered in this paper, together with the combined LiAcNeGcWcPr2i CM, referred to as 'ALL'. The latter will be described at the end of this section. Figure 7 shows that the area required for the Li, the Ac, the Ne the Gc and the


(a) ALL (LiAcNeGcWcPr2i)
Pseudo-Random Gen / Register
R3 Q R2 R1 $\overline{Q}$ R0
+1 incrementer logic
C3 O C2 C1 $\overline{O}$ C0
Ne Sequence
Li,Ac,Gc, Wc,Pr,2i Sequence
Controls
O(N-1..0)
CTRL (3):
(0): NeU/D
(1): 2iU/D, LiU/D, Wc, Gc, Pr
(2): 2iU/D, AcQ0,
CTRL (i):
(0): NeU/D
(1): AcQ0
(2): 2iU/D, LiU/D, Wc, Gc, Pr
(3): 2iU/D
CTRL (0):
(0): NeU/D
(1,2): 2iU/D
(3): 2iU/D,AcQ0
(4): 2iU/D, LiU/D, Wc, Gc, Pr
C3 $\overline{C3}$ R3 $\overline{R3}$ R0 $\overline{R0}$
Ne Li,Ac,Wc,Gc,Pr,2i
MUX3
CTRL3
O3
Ci $\overline{Ci}$ R(i+1) R(i+1) Ri $\overline{Ri}$ R0 $\overline{R0}$
Ne Li,Ac,Wc,Gc,Pr,2i
MUX(i)
CTRL(i)
O(i)
C0 $\overline{C0}$ R3 $\overline{R3}$ R2 $\overline{R2}$ R1 $\overline{R1}$ R0 $\overline{R0}$
Ne Li,Ac,Wc,Gc,Pr,2i
MUX0
CTRL(0)
O0
Fig. 8. Li, Ac, Ne, Gc, Wc, Pr and 2i AG

Wc CMs are very comparable, as also can be concluded from Figure 3, 5 and 6, as well as from Table 2, which describes the Mux-network. Note that the 2^i CM requires a larger area, because of the fact that the muxes for address bit$_i$, for $i > 0$, require two inputs pairs, while the mux for address bit$_0$ requires N input pairs, see Section 4, and Figure 5.

The area for the ALL AG is **only** 2.42 to 2.95 times the area of the Li AG, depending on the size of N (the larger N, the smaller the relative size of the ALL AG). Compared with a brute-force implementation, the area required for the ALL AG is reduced by over 60% area; e.g., for N=24, the new ALL AG consumes 1054 gates, and 3070 gates for brute-force implementation [12]. Hence, the described Up-counter – Mux-network approach results in a significant area reduction.

Figure 8 concludes this paper. It has been included to illustrate the effectiveness of the new AG implementation method. Figure 8 implements the 'ALL' AG, which supports *all* CMs described in this paper: the Li, the Ac, the Ne, the Gc, the Wc, the Pr and the 2^i CMs.

A block diagram is shown in the left-upper part of the figure. It consists of a Linear Feedback Shift Register (termed the 'Pseudo-Random Gen./Register'), with outputs R_3, R_2, R_1 and R_0; a '+1 increment logic' unit, with outputs C_3, C_2, C_1 and C_0, and a multiplexer with outputs O_3, O_2, O_1 and O_0. The multiplexer has the capability to select the *Next address* or the *Current address*, see also Figure 5(d)). This multiplexer effectively consists of N=4 multiplexers, one for each address bit. The details of these muxes is shown in the lower part of Figure 8; while their control inputs are given in the lists, located in the right-upper part of the figure.

From left-to-right, it shows the mux for Address bit 'Q_3', followed by the mux for 'Q_i', for 0 <i<N, and last, the mux for 'Q_0'. The mux with output O_3 has three pairs of inputs: C_3 and $\overline{C_3}$ feed Input '0' of the mux, R_3 and $\overline{R_3}$ feed Input '1', while R_0 and $\overline{R_0}$ feed Input '0'. The list with header 'CTRL(3)' describes the way the mux is controlled: when input pair '0' is selected, then the **Ne**xt CM, for both the **U**p and **D**own (**NeU/D**) Address Orders (AOs) is enabled. Input pair '1' selects the following CMs: the $\mathbf{2^i}$ CM, for the **U**p and **D**own AOs, the **Li**near CM for the **U**p and **D**own AOs, the **Wc** CM, the **Gc** CM, and the **Pr** CM; as indicated by '**2iU/D,LiU/D,Wc,Gc,Pr**' in the list of 'CTRL(3)'. Note that they all share the same mux inputs! The explanation of the muxes $O(i)$ and $O(0)$ is similar.

References

[1] R. Aitken, et. al, 'A Modular Wrapper Enabling High Speed BIST and Repair for Small Wide Memories', *Proc. of Int. Test Conference*, pp. 997-1005, 2004.

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

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

[4] X. Du, N. Mukherjee, W.T Cheng and S.M Reddy, 'Full-speed field-programmable memory BIST architecture', *Proc. of Int. Test Conference*, pp. 1173-1182, 2005.

[5] 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.

[6] S. Hamdioui, A. van de Goor and M. Rodgers, 'Memory Fault Modeling Trends: A Case Study', *Journal of Electronic Testing: Theory and Applications (JETTA)*, Vol. 20, Nr. 3, pp. 245-255, 2004.

[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 Asian Test Symposium*, pp. 31-36, 2009.

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

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

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

[15] 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, 25082511.

[16] ITRS2001. International Technology Roadmap for Semiconductors 2001. http://public.itrs.net/Files/20001ITRS/?Home.html

[17] A. van de Goor, C. Jung, S. Hamdioui and G.N. Gaydadjiev, 'Low-cost, Customized and Flexible SRAM MBIST Engine', *Proc. of the IEEE Int. Symp. on Design and Diagnostics of Electrronic Circuits and Systems*, 2010, pp. 382-387.

[18] P. Mazumder, 'Parallel Testing of Parametric Faults in a Three Dimensional Random-access Memory', *Proc. of the IEEE Int. Test Conf.* , pp. 933-941, 1988.

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

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

[21] Ad J. van de Goor, Said Hamdioui and Halil Kukner, 'Generic, Orthogonal and Low-cost March Element based Memory BIST', Submitted to *Int. Test Conf.*, fall 2011.