

Universal Address Sequence Generator for Memory Built-in Self-test*

Ireneusz Mrozek[†]

Bialystok University of Technology, Bialystok, Poland

i.mrozek@pb.edu.pl

Nikolai A. Shevchenko

Gymnasium, Darmstadt, Germany

nik.sh.de@gmail.com

Vyacheslav N. Yarmolik

Belarusian State University of Informatics and

Radioelectronics, Minsk, Belarus

yarmolik10ru@yahoo.com

Abstract. This paper presents the universal address sequence generator (UASG) for memory built-in-self-test. The studies are based on the proposed universal method for generating address sequences with the desired properties for multirun march memory tests. As a mathematical model, a modification of the recursive relation for quasi-random sequence generation is used. For this model, a structural diagram of the hardware implementation is given, of which the basis is a storage device for storing so-called direction numbers of the generation matrix. The form of the generation matrix determines the basic properties of the generated address sequences. The proposed UASG generates a wide spectrum of different address sequences, including the standard ones, such as linear, address complement, gray code, worst-case gate delay, 2^i , next address, and pseudorandom. Examples of the use of the proposed methods are considered. The result of the practical implementation of the UASG is presented, and the main characteristics are evaluated.

Keywords: antirandom tests, controlled random tests, multiple tests, RAM testing.

*This work was supported by the grant WZ/WI-IIT/2/2020 from Bialystok University of Technology, Faculty of Computer Science

[†]Address for correspondence: Bialystok University of Technology, Bialystok, Poland.

1. Introduction

The percentage of embedded memories in a chip is increasing. Thus, memory is a major portion of the current system-on-a-chip designs (occupying more than 70%) and is expected to rise to 95% of the area overhead [1, 2, 3, 4]. The density of modern memory is rapidly increasing compared to random logic. Additionally, the smaller feature size and increasing space occupied by the memory on a chip result in an enormous critical chip area that may potentially have defects.

Memory faults can be divided on the basis of the number of memory cells being faulty, namely into one-cell faults (e.g., stuck-at faults, stuck open faults, and transition faults), and multiple cells faults (first of all pattern sensitive faults - PSF). The first group of faults is well detectable by existing classical march tests. In the case of the second group of faults, the problem is much more difficult. Although many approaches have been proposed in the literature [5, 6, 7, 8, 9, 10, 11, 12, 13, 14], the issue of efficient detection of multiple cell faults is still open.

The traditional approach based on direct memory access for testing is costly in terms of silicon area, routing complexity, and test application time [15]. The memory built-in self-test (MBIST) has become an attractive alternative and can offer benefits, such as at-speed testing and, therefore, high fault coverage [16, 17]. Traditionally, the MBIST solutions are based on march test algorithms.

Due to the linear complexity, regularity, symmetry and simplicity of the hardware implementation, the march tests are usually a preferred and often the only reasonable method for RAM testing. March test algorithms consists of a set of march elements. March elements are a finite sequence of read and write operations applied to every cell in the memory by accessing all memory addresses in any order. The order of memory cells can be the same for all march elements or can be reversed for some march elements [6, 18].

Well known property of march tests is that for one run memory test execution there is no any special requirements for the address order, as well as for memory background [6]. For any address order and memory background the number of detectable memory faults, including multi-cell faults, will be the same and can be calculated according to the memory test detection ability. One of the constructive solutions for achieving high fault coverage especially for complex faults, as has been shown in [7, 19], is multi-run testing. The idea of multi-run tests was originally formulated in the context of transparent testing [20], and later exhaustive and pseudo-exhaustive RAM testing [21, 22, 23]. According to this idea, the same testing procedure is executed several times, each time with different initial conditions. As pointed out in many research studies [20, 21, 24, 25, 26], transparent tests are able to cover a wide range of memory faults (theoretically all faults). In this case, the test process requires multiple runs of one or more memory tests. It is obvious that the fault coverage of such testing processes depends both on the test used (including the number of its iterations) and the memory background and/or address order in each iteration of the test [26].

So, one of the key element of multi-run tests are the address sequences. As has been mentioned for one-run memory test execution, there is no any specific requirements for the address order [6]. The only restriction is that an entire set of all possible addresses has to be generated in an arbitrary order in an up and down direction. That is why a simple binary counter with an increment and a decrement by 1 mode can be used. It is another story in the case of multi-run memory tests. The high efficiency of such type of memory testing is obtained due to the detection of additional portions of the

complex memory faults. Any new run of the same memory test has to be done with the new initial conditions. Usually, this can be a new memory background or address order, or both background and address order. In this case, it is quite important to choose an appropriate set of memory addresses. For example, for two-run memory test, we have to select two different address sequences with a different address order. There is no doubt that a different subset can result in different fault coverage.

From the above perspective the key element of the MBIST is the address sequence generator (ASG), which is the most critical part of the area of MBIST implementation. The ASG designs are very different, and the area required for the ASG varies between 26% and 33% of the MBIST [27]. To detect complex and speed-related faults, the functionality of ASG should be extended and flexible [27, 28]. The ASG must generate an appropriate set of address sequences (ASs), with the desired address switching activity.

Several MBIST architectures have been proposed in the literature [17, 28, 29, 30, 31, 32, 33]. In [28, 32], attempts were put forward for proposed architectures of address generators with a low transition. It has been proven that efficient employment of the ASG architecture has considerably reduced the switching activity of MBIST [32]. The proposed approaches are based on a modified linear feedback shift register (LFSR), which generates the restricted sets of ASs.

In [34], the MBIST address generator is used to implement addresses with a significantly low area, less power, and high speed based on a set of multiplexers and counters. In this paper, a new architecture is analyzed and proposed with more advantageous properties. The implementation aspects of several ASs, including *linear*, *address complement*, and *gray code* sequences have been analyzed. The proposed investigation supports several designs for an ASG to generate only one AS, and their combination requires additional area overhead.

To reduce the MBIST power consumption to test the memory core of a system on a chip, the design proposed in [35] concentrated on just three types of ASG, namely LFSR-based, *linear*, and *gray code* ASs. A comparison with the standard solutions in terms of the area overhead and consumed power was presented and analyzed. The same power-reducing issue was investigated in [15] for only one LFSR-based AS.

In [27], the authors stated that the set of counting methods, commonly used in industry to detect different fault classes, including speed-related faults, consists of the *linear*, *address complement*, *gray code*, *worst-case gate delay*, 2^i , *next address*, and *pseudorandom* ASs. All efforts within these investigations have been concentrated on the optimization of ASG implementation. The AS properties and implementation aspects of several ASs have been considered. As a result, a novel, systematic, high-speed, low-power, and low-overhead implementation based on an up-counter and a set of multiplexers was presented [27].

The complexity of the MBIST is a major design issue because it requires a large area and limits the speed of the MBIST. In addition, the restriction on the set of ASs may reduce the efficiency of the memory-test procedures. To overcome this tradeoff, in this paper, the universal address sequence generator (UASG) is proposed and analyzed. The motivation of this work is to design an efficient universal MBIST ASG that generates sufficiently different ASs, including the standard ASs, compared with known solutions. The area overhead and speed issues, as crucial aspects of ASG implementations, are compared with the existing techniques.

2. Standard address sequences

The address sequence is a binary number system that satisfies the following property:

Property 1. A binary number system $A = a_m a_{m-1} \dots a_3 a_2 a_1$ consists of all possible 2^m binary combinations $a_m a_{m-1} \dots a_3 a_2 a_1$ formed in an arbitrary order, where $a_i \in \{0, 1\}$. Moreover, all combinations $a_m a_{m-1} \dots a_3 a_2 a_1$ occur in A only once.

It should be noted that there is the strong requirement to generate all addresses in an arbitrary order and the same sequence of addresses in an inverse order for memory test implementation.

Taking into account the Property 1, the address sequence A consists of $N = 2^m$ m -bit words as well as of m N -bit sequences $a_m a_{m-1} \dots a_1$. The classical address sequence (counter address sequence) for $m = 3$ is presented in Table 1.

Table 1. The classical address sequence for $m = 3$

| Address | $m = 3$ | | |
|---------|---------|-------|-------|
| | a_3 | a_2 | a_1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |

In the example (Table 1), we have eight addresses, namely 000, 001, 010, 011, 100, 101, 110, 111, and three bit sequences $a_3 = 00001111$, $a_2 = 00110011$ i $a_1 = 01010101$.

Now, let us formulate the general properties of the bit sequences a_i for any address sequences A . Let's start with the general property of $A = a_m a_{m-1} \dots a_3 a_2 a_1$ where $a_i \in \{0, 1\}$

Property 2. For any bit sequence a_i of an address sequence A there exist 2^{m-1} distinct binary combinations for $a_m a_{m-1} \dots a_{i+1} a_{i-1} \dots a_3 a_2 a_1$ with $a_i = 0$ and exactly the same number of combinations $a_m a_{m-1} \dots a_{i+1} a_{i-1} \dots a_3 a_2 a_1$ with $a_i = 1$.

The Property 2 allows to make the conclusion that there are 2^{m-1} '0' values and the same number 2^{m-1} '1' values for any bit sequence a_i within any binary numerical system $A = a_m a_{m-1} \dots a_3 a_2 a_1$.

Property 3. For any two-bit sequence of bits a_i and a_j $i \neq j$ of an address sequence A there are exactly 2^{m-2} all binary combinations, namely 00, 01, 10, 11 within the address sequence A .

The last property can be formulated for the general case as the following property.

Property 4. For any number of bits $r < m$ a_i, a_j, \dots, a_q within the standard address sequence A , where $i \neq j \neq \dots \neq q$, there are exactly 2^{m-r} all binary combinations, namely $00 \dots 0, 00 \dots 1, \dots, 11 \dots 1$ within the address codes $a_m a_{m-1} \dots a_3 a_2 a_1$.

For the given memory with 2^m cells, there is only one counter sequence described via classical algorithm. To increase the number of sequences with an entire set of m – bit addresses many standard solutions have been presented [32, 34, 35, 36, 37, 38, 39, 40].

Let us analyze Address bit permutation method as an example of one of AS generating methods.

For a one m -bit address sequence A , there are $m!$ sequences of addresses as a result of bit permutation. For example, in the case of a counter sequence $A = a_2 a_1$ we have only $2! = 2$ sequences, but for $A = a_3 a_2 a_1$ with $m = 3$ we can get $3! = 6$ sequences. All the mentioned sequences for $m = 2, 3$ are shown in Table 2.

Table 2. Sequences of address for $m = 2$ and $m = 3$

| $m = 2$ | | $m = 3$ | | | | | |
|-----------|-----------|---------------|---------------|---------------|---------------|---------------|---------------|
| $A\#1$ | $A\#2$ | $A\#1$ | $A\#2$ | $A\#3$ | $A\#4$ | $A\#5$ | $A\#6$ |
| $a_2 a_1$ | $a_1 a_2$ | $a_3 a_2 a_1$ | $a_3 a_1 a_2$ | $a_2 a_3 a_1$ | $a_2 a_1 a_3$ | $a_1 a_3 a_2$ | $a_1 a_2 a_3$ |
| 00 | 00 | 000 | 000 | 000 | 000 | 000 | 000 |
| | | 001 | 010 | 001 | 010 | 100 | 100 |
| 01 | 10 | 010 | 001 | 100 | 100 | 001 | 010 |
| | | 011 | 011 | 101 | 110 | 101 | 110 |
| 10 | 01 | 100 | 100 | 010 | 001 | 010 | 001 |
| | | 101 | 110 | 011 | 011 | 110 | 101 |
| 11 | 11 | 110 | 101 | 110 | 101 | 011 | 011 |
| | | 111 | 111 | 111 | 111 | 111 | 111 |

The key parameter for predicting the number $m!$ of memory address sequences depends only on the memory width m . For a large m , the value $m!$ can be approximated by Stirling's approximation:

$$r! \approx r^r e^{-r} \sqrt{2\pi r}. \quad (1)$$

In reality, it is a large number.

Let us sum up the above approach in terms of its implementation [41, 42]:

1. For real memory, this approach allows getting enormous amounts of address sequences.
2. There is substantial hardware overhead. For practical implementation, we need to use m m -input multiplexers and m m -bit registers to fix one of the address sequences out of all those possible.
3. Decreasing in the performance in terms of delay due to multiplexing of address bits.

3. Proposed method

The basic idea behind the proposed UASG is the significant expansion of the set of different ASs (with moderate hardware overhead), including the standard well-known and extensively used AS generated by the UASG. To achieve the goal, the fundamental bases of the binary vectors field are used [43].

The AS $A(n) = a_m(n) a_{m-1}(n) a_{m-2}(n) \dots a_2(n) a_1(n)$, where $a_i(n) \in \{0, 1\}$, $i \in \{1, 2, 3, \dots, m\}$, the m -dimensional binary vectors in binary space, are considered. Then, the problem of generating the desired AS can be regarded as m -dimensional binary vectors in binary space generation. The vector space consists of a set of elements $a_i(n)$ over which the binary addition operation, denoted by the XOR (\oplus) operation, is defined. The binary multiplication operation, denoted by the AND (\times) operation, is defined between an element $a_i(n)$ of the field and the vectors of the space. The set of linearly independent binary vectors, $v_i = \beta_1(i)\beta_2(i) \dots \beta_{m-1}(i)\beta_m(i)$, $i = \overline{1, m}$, generates m -dimensional binary vectors $A(n)$, which is called a basis of the m -dimensional binary vector space. The set of linearly independent vectors, $v_i = \beta_1(i)\beta_2(i) \dots \beta_{m-1}(i)\beta_m(i)$, generates m -dimensional binary vectors $A(n)$ with all linear combinations:

$$A(n) = b_1(n) \times v_1 \oplus b_2(n) \times v_2 \oplus \dots \oplus b_m(n) \times v_m, \quad (2)$$

where $B(n) = b_m(n)b_{m-1}(n)b_{m-2}(n) \dots b_2(n)b_1(n)$; $b_i(n) \in \{0, 1\}$, $i \in \{1, 2, 3, \dots, m\}$ and $n \in 2^m - 1$ is any entire binary vector set (AS) consisting from all possible 2^m binary combinations. Then, the vector space (composed of m bit vectors $A(n)$) formed according to (2) is of dimension m and consists of 2^m vectors, which is why the vectors ($A(n)$) can be used as ASs. For further investigations, the set of vectors $B(n)$ is regarded as linear ASs or simply binary up-counter sequences. The enormous variety of AS generated according to (2) primarily depends on the values of linearly independent vectors ($v_i = \beta_1(i)\beta_2(i) \dots \beta_{m-1}(i)\beta_m(i)$, $\beta_j(i) \in \{0, 1\}$, $j = \overline{1, m}$), which form the generating binary $m \times m$ matrix V . The only restriction for such a matrix (V) is the maximal rank achieved by choosing a linearly independent set of vectors v_i . The second argument extending the possibilities to generate different ASs is the vector set $B(n)$. This set consists of all possible 2^m binary vectors. Thus, any AS can be used as the vector set $B(n)$ for generation of new AS according to (2).

Brief analyses of the above-presented Relation (2), which can be used for AS generation, reveal at least two questions. The first question concerns the generation matrix V , and the second question addresses the computational complexity of the above-presented algorithm (2).

The rank of a random $m \times m$ matrix V with entries in $GF(2)$, which are independently chosen and equally likely to be 0 or 1 ($p(0) = p(1) = 0,5$), is analyzed in Kolchin's book [44]. He proved that the probability that the rank of a random $m \times m$ matrix is $m - s$ equals:

$$P(m, s) = 2^{-s^2} \left(\prod_{0 \leq i < \leq m-s-1} (1 - 2^{-(m-i)}) \times \left(\sum_{0 \leq i_1 \leq i_2 \dots i_s \leq m-s} 2^{-i_1 - i_2 - \dots - i_s} \right) \right). \quad (3)$$

In the case of $s = 0$, the probability of the full rank matrix is as follows:

$$P(m, 0) = \prod_{i=0}^{m-1} (1 - 2^{-i}). \quad (4)$$

As m becomes larger and larger, $P(m, s)$ approaches the limiting values, for example, if $s = 0$ and $m \rightarrow \infty$, then $P(m, 0) \approx 0,2887880950866$. At the same time, the expected value of the rank of the random matrix is $m - \sum_{s=0}^m sP(m, s)$. For a large value of m , this value approaches $m - 0,850179830874$ [44]. Thus, the number of ASs generated according to (2) for real values of m reaches astronomical values that are equal to more than 28.8% of the total number 2^{m^2} of possible $m \times m$ binary matrices. The procedure of AS generation based on Relation (2) for the case of $m = 4$ and the set of m linearly independent binary vectors $v_1 = 1011, v_2 = 1000, v_3 = 0101, v_4 = 1111$ forming the generated matrix V is presented in Table 3:

$$V = \begin{array}{c} \left| \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \right| = \left| \begin{array}{cccc} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{array} \right| \quad (5)$$

Table 3. Procedure of address sequence generation based on Relation (2)

| n | $B(n) = b_4(n)b_3(n)b_2(n)b_1(n)$ | $A(n) = a_4(n)a_3(n)a_2(n)a_1(n)$ | $B(n) \oplus B(n-1)$ |
|-----|-----------------------------------|-----------------------------------|----------------------|
| 0 | 0 0 0 0 | 0 0 0 0 | 1 1 1 1 |
| 1 | 0 0 0 1 | 1 0 1 1 | 0 0 0 1 |
| 2 | 0 0 1 0 | 1 0 0 0 | 0 0 1 1 |
| 3 | 0 0 1 1 | 0 0 1 1 | 0 0 0 1 |
| 4 | 0 1 0 0 | 0 1 0 1 | 0 1 1 1 |
| 5 | 0 1 0 1 | 1 1 1 0 | 0 0 0 1 |
| 6 | 0 1 1 0 | 1 1 0 1 | 0 0 1 1 |
| 7 | 0 1 1 1 | 0 1 1 0 | 0 0 0 1 |
| 8 | 1 0 0 0 | 1 1 1 1 | 1 1 1 1 |
| 9 | 1 0 0 1 | 0 1 0 0 | 0 0 0 1 |
| 10 | 1 0 1 0 | 0 1 1 1 | 0 0 1 1 |
| 11 | 1 0 1 1 | 1 1 0 0 | 0 0 0 1 |
| 12 | 1 1 0 0 | 1 0 1 0 | 0 1 1 1 |
| 13 | 1 1 0 1 | 0 0 0 1 | 0 0 0 1 |
| 14 | 1 1 1 0 | 0 0 1 0 | 0 0 1 1 |
| 15 | 1 1 1 1 | 1 0 0 1 | 0 0 0 1 |

In this case:

$$A(n) = b_1(n) \times v_1 \oplus b_2(n) \times v_2 \oplus b_3(n) \times v_3 \oplus b_4(n) \times v_4.$$

For example $A(5) = b_1(5) \times v_1 \oplus b_2(5) \times v_2 \oplus b_3(5) \times v_3 \oplus b_4(5) \times v_4 = 1 \times v_1 \oplus 0 \times v_2 \oplus 1 \times v_3 \oplus 0 \times v_4$ and finally we have $A(5) = v_1 \oplus v_3 = 1011 \oplus 0101 = 1110$.

Table 3 reveals that the number of operands for consecutive address $A(n)$ calculation according to (2) strongly depends on the number of 1s within the $B(n)$. Depending on the number of nonzero components of $B(n)$, up to m operands can obtain the value of $A(n)$. This indicates that the address

generation according to (2) is a time-consuming procedure, which may sufficiently reduce the rate of the test pattern generation.

To decrease the number of operations to only one bitwise XOR operation, Relation (2) can be transformed into a recursive relation [40, 36]:

$$A(n) = A(n-1) \oplus v_i^*; n = \overline{0, 2^m - 1}, i = \overline{1, m} \quad (6)$$

where

$$v_i^* = v_1 \oplus v_2 \oplus \dots \oplus v_i. \quad (7)$$

The main idea behind this transformation is based on the set of consecutive values of $B(n) \oplus B(n-1)$. Table 3 indicates that only the four different correction values, $v_1^*, v_2^*, v_3^*, v_4^*$, are used to obtain $A(n)$ from the previous value $A(n-1)$. For our previous example in Table 3, $v_1^* = v_1 = 1011$; $v_2^* = v_1 \oplus v_2 = 0011$; $v_3^* = v_1 \oplus v_2 \oplus v_3 = 0110$; $v_4^* = v_1 \oplus v_2 \oplus v_3 \oplus v_4 = 1001$.

For the general case, the recursive relation in (6) for the AS generation can be obtained from (2) using the new bases V^* constructed according to generation matrix V (7), which is built from the linearly independent vectors, $v_i = \beta_1(i)\beta_2(i) \dots \beta_{m-1}(i)\beta_m(i)$, $\beta_j(i) \in \{0, 1\}$, $j = \overline{1, m}$. The same relation in (7) can be used to obtain V from V^* , namely, $v_i = v_{i-1}^* \oplus v_i^*$. The value of the index i of the binary vector v_i^* , which is used as a term in Expression (6), depends on the so-called switching sequence, T_m , of the reflected gray code [37]. The binary reflected gray code, also known as the standard gray code, is the best-known gray code [37]. A characteristic property of the binary standard gray code is that the second half of the list of codewords can be obtained from the first half by reflection (i.e., by writing the first half backwards and replacing the first 0 with 1). Any reflected gray code is described by the switching sequence T_m . For example, for $m = 4$, this sequence has the form $T_4 = 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1$. Formally, the switching sequence T_m defines the index i of an inverted bit to obtain the new value $B(n)_g$ from the previous value $B(n-1)_g$. Index g of the $B(n)_g$ indicates the representation in the gray code of the initial binary code $B(n) = b_m(n)b_{m-1}(n)b_{m-2}(n) \dots b_2(n)b_1(n)$. The vector $B(n)_g$ in the gray code $B(n)_g = g_m(n)g_{m-1}(n)g_{m-2}(n) \dots g_2(n)g_1(n)$ can be obtained according to the following well-known relation [37]:

$$\begin{aligned} g_m(n) &= b_m(n) \\ g_i(n) &= b_{i+1}(n) \oplus b_i(n); i = \overline{1, m-1}. \end{aligned} \quad (8)$$

The values of the bits of gray code $B(n)_g$ for $m = 4$ are determined in accordance with the relations $g_4(n) = b_4(n)$, $g_3(n) = b_4(n) \oplus b_3(n)$, $g_2(n) = b_3(n) \oplus b_2(n)$, and $g_1(n) = b_2(n) \oplus b_1(n)$.

4. Address Sequence Generator

The general structure of the proposed ASG consists of three sequentially connected function blocks, as presented in Fig. 1. The first block, the switching sequence generator (SSG), is used to select one out of m vectors v_i per clock (Clk) according to the required order. As demonstrated in the previous section, the main block of the ASG is a memory block for storing m linearly independent vectors $v_i = \beta_1(i)\beta_2(i) \dots \beta_{m-1}(i)\beta_m(i)$, $\beta_j(i) \in \{0, 1\}$, $j = \overline{1, m}$, which form the binary $m \times m$

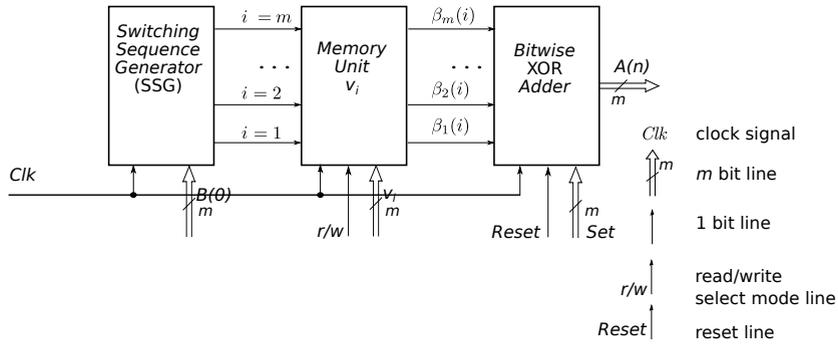


Figure 1. General structure of the proposed address sequence generator.

generation matrix V . Then, the second block is the *memory unit* consisting of m m -bit cells for storing vectors v_i . Memory units must perform *read* (r) and *write* (w) operations (r/w) for the generation of vectors v_i and upload the new values (Fig. 1). The last block is the *bitwise XOR adder* to perform the operation $A(n) = A(n - 1) \oplus v_i$. It consists of m synchronous D -type flip-flops and m two-input XOR gates. The adder moves to the next state after the clock pulse (Clk) generation. Set and reset inputs in D -type flip-flops load the all-zero state $A(0)$ (*reset*) to the adder or any other initial state $A(0)$ (*set*). At the output of the adder, the desired AS is generated.

The structure of the second (memory unit) and third (bitwise XOR adder) blocks is quite simple and standard, but the construction of the first block is not as obvious. The architecture of this block, namely, the SSG proposed in this paper, is characterized in Fig. 2.

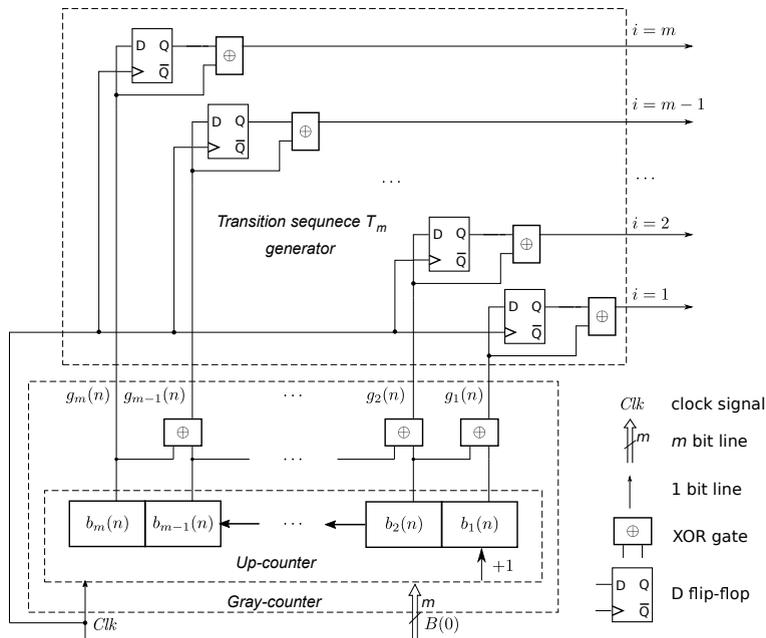


Figure 2. Switching sequence T_m generator.

The main block of the ASG is the SSG, T_m , which determines the sequence of the selection of the vectors v_i of the matrix V (2). As noted, in each cycle of operation, only one output of the SSG generates an enable signal that determines the selected vector v_i by its index i .

The *up-counter* is the main part of the SSG, which consists of the m -bit binary counter that performs two micro-operations. The first operation is the *increment by 1 (+1)* operation to make the transition from $B(n-1)$ to $B(n)$. The second operation is the *load* instruction, which is used to upload the initial state $B(0)$ to the up-counter. Both operations are synchronous and are performed by the clock signal Clk . The *gray counter* in Fig. 2 consists of an up-counter and $m-1$ two-input XOR gates connected according to (8). The SSG T_m performs the bitwise XOR operation between consecutive vectors $B(n-1)_g$ and $B(n)_g$ to obtain one out of m output selection vector v_i signals. It consists of m D -type flip-flops and m two-input XOR gates.

The proposed UASG illustrated in Figs. 1 and 2 generates any AS depending on the values of the generated matrix V . The only restriction for V is the linear independence of the binary vectors v_i . For example, for $m = 4$ and the matrix (5) the procedure for obtaining the AS (set of all possible m bit vectors $A(n)$) in detail is presented in Table 4.

Table 4. Procedure of AS generation based on relation (6)

| n | $B(n)$ | $B(n)_g$ | $B(n)_g \oplus B(n-1)_g$ | T_m | v_i | $\uparrow A(n)$ | $\downarrow A(n)$ | $\uparrow A^*(n)$ |
|-----|---------|----------|--------------------------|-------|---------|-----------------|-------------------|-------------------|
| 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 4 | 1 1 1 1 | 0 0 0 0 | 1 1 1 1 | 1 0 0 0 |
| 1 | 0 0 0 1 | 0 0 0 1 | 0 0 0 1 | 1 | 1 0 1 1 | 1 0 1 1 | 0 1 0 0 | 0 0 1 1 |
| 2 | 0 0 1 0 | 0 0 1 1 | 0 0 1 0 | 2 | 1 0 0 0 | 0 0 1 1 | 1 1 0 0 | 1 0 1 1 |
| 3 | 0 0 1 1 | 0 0 1 0 | 0 0 0 1 | 1 | 1 0 1 1 | 1 0 0 0 | 0 1 1 1 | 0 0 0 0 |
| 4 | 0 1 0 0 | 0 1 1 0 | 0 1 0 0 | 3 | 0 1 0 1 | 1 1 0 1 | 0 0 1 0 | 0 1 0 1 |
| 5 | 0 1 0 1 | 0 1 1 1 | 0 0 0 1 | 1 | 1 0 1 1 | 0 1 1 0 | 1 0 0 1 | 1 1 1 0 |
| 6 | 0 1 1 0 | 0 1 0 1 | 0 0 1 0 | 2 | 1 0 0 0 | 1 1 1 0 | 0 0 0 1 | 0 1 1 0 |
| 7 | 0 1 1 1 | 0 1 0 0 | 0 0 0 1 | 1 | 1 0 1 1 | 0 1 0 1 | 1 0 1 0 | 1 1 0 1 |
| 8 | 1 0 0 0 | 1 1 0 0 | 1 0 0 0 | 4 | 1 1 1 1 | 1 0 1 0 | 0 1 0 1 | 0 0 1 0 |
| 9 | 1 0 0 1 | 1 1 0 1 | 0 0 0 1 | 1 | 1 0 1 1 | 0 0 0 1 | 1 1 1 0 | 1 0 0 1 |
| 10 | 1 0 1 0 | 1 1 1 1 | 0 0 1 0 | 2 | 1 0 0 0 | 1 0 0 1 | 0 1 1 0 | 0 0 0 1 |
| 11 | 1 0 1 1 | 1 1 1 0 | 0 0 0 1 | 1 | 1 0 1 1 | 0 0 1 0 | 1 1 0 1 | 1 0 1 0 |
| 12 | 1 1 0 0 | 1 0 1 0 | 0 1 0 0 | 3 | 0 1 0 1 | 0 1 1 1 | 1 0 0 0 | 1 1 1 1 |
| 13 | 1 1 0 1 | 1 0 1 1 | 0 0 0 1 | 1 | 1 0 1 1 | 1 1 0 0 | 0 0 1 1 | 0 1 0 0 |
| 14 | 1 1 1 0 | 1 0 0 1 | 0 0 1 0 | 2 | 1 0 0 0 | 0 1 0 0 | 1 0 1 1 | 1 1 0 0 |
| 15 | 1 1 1 1 | 1 0 0 0 | 0 0 0 1 | 1 | 1 0 1 1 | 1 1 1 1 | 0 0 0 0 | 0 1 1 1 |

The first column contains the binary values $B(n) = b_4(n)b_3(n)b_2(n)b_1(n)$ of the *up-counter* starting from the all-zero state. Transformed into gray code, $B(n)_g = g_4(n)g_3(n)g_2(n)g_1(n)$ vectors as the output sequence of the *gray counter* are listed in the next column (Fig 2 and Table 4). Columns $B(n)_g \oplus B(n-1)_g$, T_m and v_i contain the transition sequence output signals used for selecting one $v_i = \beta_1(i)\beta_2(i)\beta_3(i)\beta_4(i)$ out of the four vectors (v_1, v_2, v_3, v_4) of the matrix (5) as well as corresponding v_i vectors. The output AS (vectors $A(n) = A(n-1) \oplus v_i; n = 0, 1, 2, \dots, 2^4 - 1, i \in \{1, 2, 3, 4\}$ in the next column $\uparrow A(n)$ can be regarded as the up-sequence. The initial value for the up-sequence generation of the all-zero vector $A(0) = 0000$ was chosen. The corresponding down-sequence, the sequence with the reversed address order, is presented in column $\downarrow A(n)$. To generate the down-sequence, the initial value of $\downarrow A(0)$ must be equal to the last $\uparrow A(15)$ value of the up-sequence. In this case, $\downarrow A(0) = \uparrow A(15) = 1111$ (Table 4).

To summarize, the initial value $\Downarrow A(0)$ for the down-sequence generation must be equal to the final state $\Uparrow A(n-1)$ of the up-sequence, which follows from the properties of the XOR operation and is formulated as Statement 1.

Statement 1. A decreasing sequence (down-sequence) of addresses $\Downarrow A(n), n \in \{0, 1, 2, \dots, 2^m-1\}$ with respect to the increasing sequence (up-sequence) of addresses $\Uparrow A(n), n \in \{0, 1, 2, \dots, 2^m-1\}$, for which $\Downarrow A(n) = \Uparrow A(2^m-1-n)$, is generated using Relation (6) and the same generator matrix V as for generating $\Uparrow A(n)$ with the starting address $\Downarrow A(0)$ equal to $\Uparrow A(2^m-1)$.

If the initial state $A(0)$ is not an all-zero state, the AS differs from the original AS obtained for the zero starting state. All bits of vector $A(n) = a_m(n)a_{m-1}(n)a_{m-2}(n) \dots a_2(n)a_1(n)$ for which $a_j(0) = 1, j \in \{1, 2, 3, \dots, m\}$ are inverted. Table 4 reveals the sequence $\Uparrow A^*(n)$ for which $A^*(0) = 1000$ is the copy of $\Uparrow A(n)$ with just the fourth bit inverted. The sequence $\Downarrow A(n)$ was obtained from the inverted values of $\Uparrow A(n)$ because $\Downarrow A(0) = 1111$.

5. Address sequences

The declared goal of the presented research is a UASG with a wide spectrum of generated sequences. The general expression for AS generation according to the algorithm implemented as the UASG is based on the recursive relation in (6) and has the following form:

$$\begin{aligned} A(0) &= A; \\ A(n) &= A(n-1) \oplus v_{i(T_m(B))} \\ & \quad n = \overline{1, 2^m-1}; i = \overline{1, m} \end{aligned} \quad (9)$$

The initial conditions of UASG depend on the values of two constants $A = a_m a_{m-1} \dots a_3 a_2 a_1$ and $B = b_m b_{m-1} \dots b_3 b_2 b_1$, where $a_i, b_i, \in \{0, 1\}$, and m m -bit binary vectors $v_i = \beta_1(i)\beta_2(i) \dots \beta_{m-1}(i)\beta_m(i), \beta_j(i) \in \{0, 1\}, j = \overline{1, m}$, which form the binary $m \times m$ generation matrix V . Constants A and B are represented by the initial states of *bitwise XOR adder* and *up-counter*, respectively (Figs. 1 and 2). Concerning the constants A and B , no restrictions exist for their values and, usually, their standard meaning is all-zero values. For some UASG implementations, the zero initial conditions for A and B can save the required area for the fabrication of the generator.

The only requirement for the generation matrix V is its maximal rank, which generates cyclic ASs with a length of 2^m [43]. The required order of the vectors $v_i(T_m(B))$ depends on the switching sequence $T_m(B)$ of the reflected gray code generated by the *gray counter* (Fig. 2). The gray counter is constructed on the bases of the binary *up-counter*, which can use any initial state $B(0)$. Usually, the starting value of the *up-counter* is a zero binary vector, and the sequence of the used vectors v_i corresponds to the standard reflected gray code sequence T_m . Applying nonzero values of $B(0) \neq 000 \dots 0$ initiates the generation process of the shifted version of $T_m = T_m(B)$, where B determines the number of shifts. Then, $i = i(T_m(B))$ is the function of $T_m(B)$, which defines the sequence of the selected vectors for AS generation.

The analysis of the existing memory tests reveals that it is necessary to generate addresses in the reverse sequence $\Downarrow A(n)$ concerning the original $\Uparrow A(n)$ ASs and their various modifications. To solve

this problem, the peculiar properties of the proposed model (9) for AS generation can be used for further investigation. The equivalence of the addition (XOR) and subtraction operations by modulo two (\oplus) [43] and the symmetry of the switching sequence T_m of the reflected gray code [37] are basic features of (9). Within the framework of the proposed model in (9) of generating ASs, the formation of a sequence of decreasing $\downarrow A(n)$ addresses concerning $\uparrow A(n)$ is consistent with Statement 1 [45]. Statement 1 is true for any AS generated according to ((9)) and the arbitrary initial value of $B(0)$. For all-zero values of $B(0)$, the example of up and down ASs is given in Table 4.

Statement 2. A shifted copy of the AS by any l number of positions compared with the original AS for the case of all-zero values of $B(0)$ and $A(0)$ generated according to (9) is obtained when $B(0) = l$ and $A(0) = A(l)$.

Statement 2 also is true for any AS generated based on the chosen mathematical model (9) and generates the shifted version of the address order, which is very important for multirun memory testing [38, 46]. As an example, the shifted copy of AS by $l = 3$ positions is presented in Table 5. For this case, $m = 4$ and the same matrix V (5) is used.

Table 5. Procedure of shifted AS generation by UASG according to relation (9)

| n | $A(n)$ with $B(0) = 0000$, $A(0) = 0000$ | | | $A(n)$ with $B(0) = 0011$, $A(0) = 0000$ | | | $A(n)$ with $B(0) = 0011$, $A(0) = 1000$ | | |
|-----|--|----------|---------|--|----------|---------|--|---------|--|
| | $B(n)$ | $T_m(B)$ | $A(n)$ | $B(n)$ | $T_m(B)$ | $A(n)$ | $T_m(B)$ | $A(n)$ | |
| 0 | 0 0 0 0 | $i = 4$ | 0 0 0 0 | 0 0 1 1 | $i = 1$ | 0 0 0 0 | $i = 1$ | 1 0 0 0 | |
| 1 | 0 0 0 1 | $i = 1$ | 1 0 1 1 | 0 1 0 0 | $i = 3$ | 0 1 0 1 | $i = 3$ | 1 1 0 1 | |
| 2 | 0 0 1 0 | $i = 2$ | 0 0 1 1 | 0 1 0 1 | $i = 1$ | 1 1 1 0 | $i = 1$ | 0 1 1 0 | |
| 3 | 0 0 1 1 | $i = 1$ | 1 0 0 0 | 0 1 1 0 | $i = 2$ | 0 1 1 0 | $i = 2$ | 1 1 1 0 | |
| 4 | 0 1 0 0 | $i = 3$ | 1 1 0 1 | 0 1 1 1 | $i = 1$ | 1 1 0 1 | $i = 1$ | 0 1 0 1 | |
| 5 | 0 1 0 1 | $i = 1$ | 0 1 1 0 | 1 0 0 0 | $i = 4$ | 0 0 1 0 | $i = 4$ | 1 0 1 0 | |
| 6 | 0 1 1 0 | $i = 2$ | 1 1 1 0 | 1 0 0 1 | $i = 1$ | 1 0 0 1 | $i = 1$ | 0 0 0 1 | |
| 7 | 0 1 1 1 | $i = 1$ | 0 1 0 1 | 1 0 1 0 | $i = 2$ | 0 0 0 1 | $i = 2$ | 1 0 0 1 | |
| 8 | 1 0 0 0 | $i = 4$ | 1 0 1 0 | 1 0 1 1 | $i = 1$ | 1 0 1 0 | $i = 1$ | 0 0 1 0 | |
| 9 | 1 0 0 1 | $i = 1$ | 0 0 0 1 | 1 1 0 0 | $i = 3$ | 1 1 1 1 | $i = 3$ | 0 1 1 1 | |
| 10 | 1 0 1 0 | $i = 2$ | 1 0 0 1 | 1 1 0 1 | $i = 1$ | 0 1 0 0 | $i = 1$ | 1 1 0 0 | |
| 11 | 1 0 1 1 | $i = 1$ | 0 0 1 0 | 1 1 1 0 | $i = 2$ | 1 1 0 0 | $i = 2$ | 0 1 0 0 | |
| 12 | 1 1 0 0 | $i = 3$ | 0 1 1 1 | 1 1 1 1 | $i = 1$ | 0 1 1 1 | $i = 1$ | 1 1 1 1 | |
| 13 | 1 1 0 1 | $i = 1$ | 1 1 0 0 | 0 0 0 0 | $i = 4$ | 1 0 0 0 | $i = 4$ | 0 0 0 0 | |
| 14 | 1 1 1 0 | $i = 2$ | 0 1 0 0 | 0 0 0 1 | $i = 1$ | 0 0 1 1 | $i = 1$ | 1 0 1 1 | |
| 15 | 1 1 1 1 | $i = 1$ | 1 1 1 1 | 0 0 1 0 | $i = 2$ | 1 0 1 1 | $i = 2$ | 0 0 1 1 | |

Table 5 reveals that the AS $A(n)$ that is shifted by $l = 3$ positions is generated for $B(0) = 3_{(10)} = 0011_{(2)}$ and $A(3) = 1000$. For real values of l , the only problem is to determine the meaning of $A(l)$, which requires additional calculations. These calculations are based on the following statement, which follows from the above-presented discussions concerning the binary vector space [43, 44].

Statement 3. The AS, which is generated as m -dimensional binary vectors according to (2) based on generating an $m \times m$ matrix V with a maximal rank, can be obtained from the recursive relation in (6) with the generation matrix V^* received from (7) and vice versa.

From this statement, it follows that, if the AS $A(n)$ that is presented in Table 5 is generated according to Relation (6) or (9) using the matrix in (5), then the same sequence $A(n)$ is generated based on (2) for the generation matrix V^* , which is calculated as follows:

$$V^* = \begin{vmatrix} v_1^* \\ v_2^* \\ v_3^* \\ v_4^* \end{vmatrix} = \begin{vmatrix} v_1 \\ v_1 \oplus v_2 \\ v_2 \oplus v_3 \\ v_3 \oplus v_4 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix} \quad (10)$$

Applying Relation (2) and using the matrix in (10), $A(3) = v_1^* \oplus v_2^* = 1011 \oplus 0011 = 1000$.

Applying different values of $B(0)$ and $A(0)$ generates a wide spectrum of different ASs. Among which, sequences of addresses $A(n) = a_m(n)a_{m-1}(n)a_{m-2}(n)\dots a_2(n)a_1(n)$ with inverted bits $a_i(n), i \in \{1, 2, 3, \dots, m\}$ have actively been used in practice [38, 47, 39]. Considering that the bit inversion $\bar{a}_i(n)$ is equivalent to the XOR operation $\bar{a}_i(n) = a_i(n) \oplus 1$, the desired set of inverted bits within $A(n)$ can be specified by the initial value $A(0)$. The examples of such ASs are illustrated in Table 5. This technique, based on just setting a nonzero initial value $A(0)$, allows to generate $2^m - 1$ different sequences, where m is the dimension of the vector space described by the generation matrix V .

6. Most common Address Sequences for memory built-in self-test

The generalized mathematical model (9) presented in the previous section is an extension of the mathematical model used for binary vector generation. The basis of this model is in the form of the generation matrix V , which determines the main properties of ASs and identifies their subsets. For memory testing, the address generator must generate several ASs because each sequence and the combinations of them have their properties that are closely related to the memory-test fault-detection capability [27, 48, 49]. The generation of the most important and quite common ASs listed in [27] based on UASG is considered next.

Linear ASs, also called *counting ASs* are the first in the set of an AS family. For the formation of *counting (counter)* sequences formed by binary counting circuits (counters), it is necessary to form a generation matrix V following Statement 4.

Statement 4. The *linear(counting)* AS is generated by UASG (9) when the generation matrix V is the lower triangular matrix relative to the antidiagonal with only nonzero (1s) entries on and below the antidiagonal

An example of such an AS is presented in Table 6 for the case of $m = 4$ and $B(0) = A(0) = 0000$. The linear AS belongs to the set of 2^j ASs, which generates all address pairs with a Hamming distance equal to 1 [27]. The linear AS is the 2^j AS with $j = 0$, obtaining the address order incremented by 1. The complete set of 2^j AS where $j \in \{0, 1, 2, \dots, m - 1\}$, can be generated based on the proposed solution according to the next statement.

Statement 5. The 2^j AS is generated using UASG (9) when the generation matrix V is the matrix obtained as the column permutation of the lower triangular matrix relative to the antidiagonal with only nonzero ($1s$) entries on and below the antidiagonal with all $1s$ in the $m - j$ column.

Table 6 contains an example of $2^j = 4$ AS for $j = 2$, generating the addresses incremented by 4. The generation matrix V has all an $1s$ column with the index $m - j = 4 - 2 = 2$. In the previous section, the up and down (increasing/decreasing) sequence generation techniques for the case of any type of AS were presented. Applying the technique described by Statement 1, it is easy to generate decreasing order of the 2^j AS. The example in Table 6 repeats the example presented in [27], for which the switching activity of the other bits of $A(n) = a_4(n)a_3(n)a_2(n)a_1(n)$, except the $(j + 1)$ th bit corresponding to all the $1s$ $m - j$ column, is constant. The proposed solutions implemented as the UASG set any switching activity for all bits of $A(n)$, starting from the minimal $2^0 = 1$ up to the maximal 2^{m-1} [36].

Table 6. Most-used address sequence generation using the universal address sequence generator

| n | Linear | $2^j = 4$ | Complement | Limited | Gray Code | Random |
|-----|---------|-----------|------------|---------|-----------|---------|
| | 0 0 0 1 | 0 1 0 0 | 1 1 1 1 | 1 1 1 1 | 0 0 0 1 | 1 0 0 0 |
| | 0 0 1 1 | 1 1 0 0 | 1 1 1 0 | 1 1 1 0 | 0 0 1 0 | 1 1 0 0 |
| | 0 1 1 1 | 1 1 0 1 | 1 1 0 0 | 1 1 0 1 | 0 1 0 0 | 1 1 1 0 |
| | 1 1 1 1 | 1 1 1 1 | 1 0 0 0 | 1 0 1 1 | 1 0 0 0 | 1 1 1 1 |
| 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 1 0 0 0 |
| 1 | 0 0 0 1 | 0 1 0 0 | 1 1 1 1 | 1 1 1 1 | 0 0 0 1 | 0 0 0 0 |
| 2 | 0 0 1 0 | 1 0 0 0 | 0 0 0 1 | 0 0 0 1 | 0 0 1 1 | 1 1 0 0 |
| 3 | 0 0 1 1 | 1 1 0 0 | 1 1 1 0 | 1 1 1 0 | 0 0 1 0 | 0 1 0 0 |
| 4 | 0 1 0 0 | 0 0 0 1 | 0 0 1 0 | 0 0 1 1 | 0 1 1 0 | 1 0 1 0 |
| 5 | 0 1 0 1 | 0 1 0 1 | 1 1 0 1 | 1 1 0 0 | 0 1 1 1 | 0 0 1 0 |
| 6 | 0 1 1 0 | 1 0 0 1 | 0 0 1 1 | 0 0 1 0 | 0 1 0 1 | 1 1 1 0 |
| 7 | 0 1 1 1 | 1 1 0 1 | 1 1 0 0 | 1 1 0 1 | 0 1 0 0 | 0 1 1 0 |
| 8 | 1 0 0 0 | 0 0 1 0 | 0 1 0 0 | 0 1 1 0 | 1 1 0 0 | 1 0 0 1 |
| 9 | 1 0 0 1 | 0 1 1 0 | 1 0 1 1 | 1 0 0 1 | 1 1 0 1 | 0 0 0 1 |
| 10 | 1 0 1 0 | 1 0 1 0 | 0 1 0 1 | 0 1 1 1 | 1 1 1 1 | 1 1 0 1 |
| 11 | 1 0 1 1 | 1 1 1 0 | 1 0 1 0 | 1 0 0 0 | 1 1 1 0 | 0 1 0 1 |
| 12 | 1 1 0 0 | 0 0 1 1 | 0 1 1 0 | 0 1 0 1 | 1 0 1 0 | 1 0 1 1 |
| 13 | 1 1 0 1 | 0 1 1 1 | 1 0 0 1 | 1 0 1 0 | 1 0 1 1 | 0 0 1 1 |
| 14 | 1 1 1 0 | 1 0 1 1 | 0 1 1 1 | 0 1 0 0 | 1 0 0 1 | 1 1 1 1 |
| 15 | 1 1 1 1 | 1 1 1 1 | 1 0 0 0 | 1 0 1 1 | 1 0 0 0 | 0 1 1 1 |

The *complement* AS described in [27] specifies the sequence that, in the even cycle, represents the linear up-sequence and, in the odd cycle, takes the complementary value of the preceding even cycle.

Statement 6. The *complement* AS is generated by the UASG (9) when the generation matrix V is the upper triangular matrix relative to the antidiagonal with only nonzero ($1s$) entries on and above the antidiagonal.

For the case of $m = 4$, Table 6 contains the set of complement addresses that takes complementary values from the odd cycle of the previous address obtained from the even cycle. In the even cycle, (Table 6) this sequence corresponds to the linear AS. The *complement* AS is extremely useful for

providing the stress behavior of a memory address decoder. In this case, the high rate of switching activity of the address bits creates considerable noise, a high level of power consumption, and maximal delay [27].

The same high efficiency can be obtained in terms of the speed-related memory faults using an AS with *limited* switching activity [45]. This type of AS obtains addresses with the highest possible switching activity. To generate such ASs, the generation matrix V should satisfy the following statement.

Statement 7. The generation matrix V of UASG (9) for the AS with *limited* (highest possible) switching activity consists of one unit column and $m-1$ columns that are different from each other, containing a 0 value in each of the $m-1$ rows, except the first row.

For the case of $m = 4$, Table 6 contains the set of addresses with *limited* activity that takes reflected grey codes in the even cycles and complementary values in the odd cycle of the previous address obtained in the even cycle (Table 6). Like the *complement* AS, the *limited* AS is also especially useful for providing the stress behavior of the memory address decoder.

To minimize the stress during memory testing, sequences with minimal switching activity are used, among those in the first place is the *gray code* AS. In the general case, the AS with the minimum switching activity (minimum Hamming distance) formed according to (9) is provided by the matrix V with a minimum number of nonzero values. For an arbitrary case, such a matrix is constructed according to Statement 8.

Statement 8. The generation matrix V for the AS generation based on (9) with minimum switching activity consists of m rows that are different from each other, each of which contains just one value of 1.

According to the above statement, such a generation matrix V characterized by a set of columns differing from each other, containing one nonzero value, as listed in Table 6 for the case of $m = 4$. This example is the standard reflected gray code sequence. In addition, $m!$ different gray code ASs exist, which can be reproduced by the UASG. In the case of $m = 4$, this number equals $4! = 24$ as a result of all permutations of the matrix V columns resulting in the bit rearrangement of the address $A(n)$.

All the above-described ASs belong to the set of so-called deterministic sequences that are widely used for MBIST. The next widely used set of ASs for memory testing involves so-called *pseudorandom* sequences that are sequences of nonrandom numbers that have properties of random sequences. The M -sequences generated by LFSR are often used as ASs [27, 32, 34, 35]. The *quasi-random* sequences also belong to the family of sequences which, being deterministic, have the main properties of random sequences [40, 45, 32, 50]. To have a similar computation overhead to pseudorandom testing, quasi-random testing uses quasi-random sequences to generate low-discrepancy and low-dispersion test cases that help deliver high fault-detection effectiveness [50].

The mathematical model described by Relation (9) and matrix V of directed numbers (m -bit binary vectors) in the form of a lower triangular matrix with a unit diagonal can be used for the case of ASs related to quasi-random sequence generation. In the general case, any square matrix V with the properties described in Statement 9 can be used for quasi-random AS generation.

Statement 9. The generation matrix V for quasi-random AS generation based on (9) has the form of a lower triangular matrix with all 1s on the main diagonal.

The specific values of the binary vectors of the lower triangular matrix correspond to the modified directed numbers that specify a specific form of the quasi-random sequence. For example, in the case in which all entries below and on the main diagonal are 1s, the generated AS is a *van der Corput* sequence [36, 45, 50]. For the case of $m = 4$, this sequence is presented in Table 6.

7. Hardware implementation

The hardware overhead of the proposed solution for the ASIC circuitry can be estimated based on the general structure of the proposed address sequence generator (ASG) in Fig. 1. The UASG consists of three main blocks as depicted in Section 3, namely the switching sequence generator (SSG), memory unit, and bitwise XOR adder. For the general case with m -bit addresses, the hardware overhead is needed for all these blocks, and the whole UASG is shown in the next table.

Table 7. Hardware overhead for UASG implementation

| Standard elements | D-type flip-flops | 2 XOR gates | Memory | Up-Counter |
|-------------------|-------------------|-------------|---------------------------|---------------------|
| SSG | m | $2m - 1$ | – | m -bit up-counter |
| Memory unit | | | m m -bit memory cells | |
| XOR adder | m | m | | |
| UASG | $2m$ | $3m - 1$ | m m -bit memory cells | m -bit up-counter |

Compared with the known solutions, especially the best solution [27], the proposed ASG requires moderate hardware overhead. According to the summarized data in Table 7, only roughly $3m$ D-type flip-flops, $3m$ 2-XOR gates, and memory with m cells each of m -bit size are needed for UASG implementations. Depending of the used technology the area overhead for UASG implementation will vary sufficiently and will be comparable with all known solutions.

The performance of the proposed generator depends only on up-counter delays because this block is the slowest one (see Fig. 2). The signal delays on the bitwise XOR adder and transition sequence T_m generator are the same and equals to the delay on D-type flip-flop and XOR gate (see Fig 1 and Fig. 2). As well as the delay on the memory unit can be estimated as the delay for read operation by the so-called memory cycle. This delay can be measured by the delay on only one D-type flip-flop in a case of register type of the memory. In a worse case the total delay of UASG will includes delay on up-counter, delay on three D-type flip-flops and delay on three XOR gates. Taking into account that up-counter consists of m sequentially connected D-type flip-flops its delay time will be dominated one for the real applications when m usually greater than 32. As the conclusion of this discussions can be stated that frequency of formation of synchronizing signals (Clk) in the proposed UASG generator will be the same as in most similar generators utilized binary counter.

The best known solution allows generating only seven types of address sequences [27]. Additional options in the existing solutions for new address sequence generation require additional hardware,

decreasing its rough performance. In the case of UASG, any possible address sequence can be generated without additional hardware with the same performance.

The main characteristics of the ASG were investigated using its implementation on FPGA, Intel Cyclone V (5CSXFC6D6F31C8ES), illustrated in Fig. 3. The specified FPGA consists of 41910 adaptive logic modules and 553 SRAM memory blocks (M10k). The generator implementation for $m = 8$ required 17 adaptive logic models and one M10k internal memory unit, which is less than 1% of the FPGA chip area. The timing parameters of the generator correspond to the maximum possible timing parameters of the FPGA.

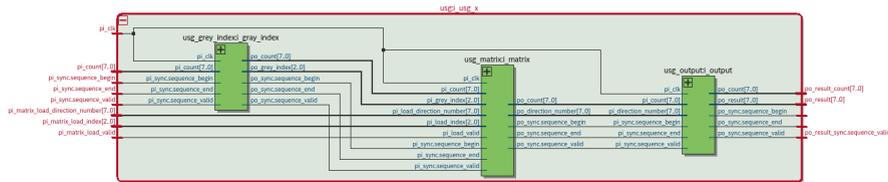


Figure 3. Implementation of an address sequence generator on FPGA.

The implementation of the ASG in Fig. 3 completely corresponds to the structure given earlier (Fig. 1). The input, output, and intermediate nodes of the implemented device (Fig. 3) and its detailed structure (Fig. 1) and descriptions are in full compliance. The examples of the generation of different ASs are illustrated in Fig. 4.

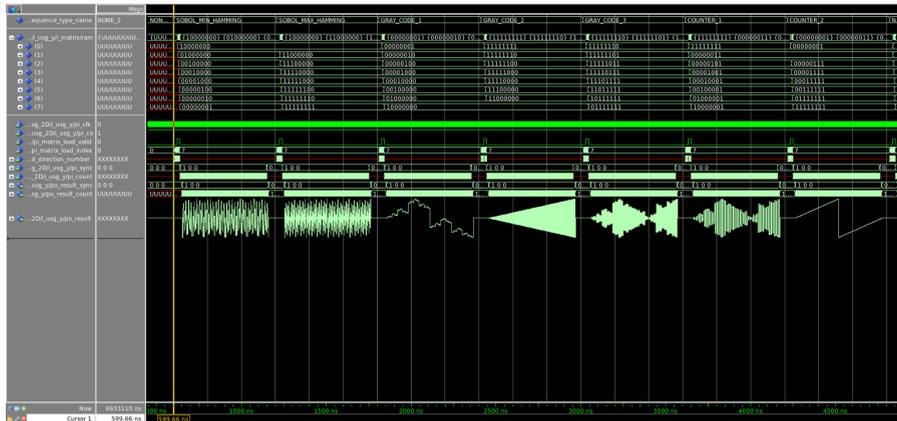


Figure 4. Waveform from the FPGA implementation of the address generator.

Figure 4 displays the ModelSim simulation result waveform from the FPGA implementation of the generator for $m = 8$. The top row reveals the type of sequence that is generated by the UASG. The eight rows below indicate the entries in the generation matrix V . The content of the matrix is loaded with the matrix load valid signal. The UASG then generates the AS according to the loaded matrix. The bottom row is a visualization of the generated sequences. The depicted waveforms are

created from the values of the ASG output. The displayed sequences are (listed from left to right) the Sobol sequence with minimal Hamming distance, the Sobol sequence with the maximum Hamming distance, three types of gray code, and two types of counters, including a linear counter.

The power consumption of UASG (Fig. 3) was analyzed using Quartus Prime (v. 19.1.0 Build 670 09/22/2019 SJ Lite Edition). The results of the analysis are given in Table 8, which indicates the minimum power consumption of the proposed device. The UASG time parameters correspond to the maximum possible FPGA time parameters.

Table 8. Power consumption analysis results

| | |
|--|-----------|
| Total Thermal Power Dissipation | 463.45 mW |
| Core Dynamic Thermal Power Dissipation | 14.63 mW |
| Core Static Thermal Power Dissipation | 415.27 mW |
| I/O Thermal Power Dissipation | 33.56 mW |

8. Conclusion

The use of a modified mathematical model of quasi-random sequence generation expanded the capabilities of the ASG in terms of a significant increase in the number of types of such sequences. The essence of the method consists of the synthesis of the required generation matrix of maximum rank, providing the given values of switching activity. The limitations of the proposed technique are discussed, which are associated with the possible conflicting requirements for the values of the weights of the rows of the matrix and their linear independence. Examples of the use of such sequences for MBIST design are provided. A practical implementation of the ASG is presented, demonstrating the feasibility of such a device with minimal hardware costs and maximum speed.

References

- [1] Test and Test Equipment. The International Technology Roadmap for Semiconductors, <http://www.itrsnet/>. 2015.
- [2] Bhunia S, Tehranipour M. System on Chip (SoC) Design and Test. In: Hardware Security. United States: Morgan Kaufmann Publishers; 2019. p. 47–78.
- [3] Marinissen EJ, Prince B, Keltel-Schulz D, Zorian Y. Challenges in embedded memory design and test. In: Proceedings of the Design, Automation and Test in Europe. vol. 2; 2005. p. 722–727. doi:10.1109/DATE.2005.92.
- [4] Bushnell M, Agrawa DV. Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits. Frontiers in Electronic Testing. Springer US; 2002. doi:10.1007/b117406.
- [5] Hayes JP. Detection of Pattern-Sensitive Faults in Random-Access Memories. IEEE Transactions on Computers. 1975;24(2):150–157.

- [6] van de Goor AJ. Testing Semiconductor Memories: Theory and Practice. Chichester, England: John Wiley & Sons; 1991. ISBN: 0471925861, 9780471925866.
- [7] Cockburn BF. Deterministic tests for detecting scrambled pattern-sensitive faults in RAMs. In: Proceedings of the IEEE International Workshop on Memory Technology, Design and Testing. IEEE Computer Society; 1995. p. 117–122.
- [8] Franklin M, Saluja KK. Testing reconfigured RAM's and scrambled address RAM's for pattern sensitive faults. IEEE Transactions on CAD of Integrated Circuits and Systems. 1996;15(9):1081–1087.
- [9] Cheng KL, Wu CW. Neighborhood Pattern-Sensitive Fault Testing for Semiconductor Memories. In: Proceedings of the VLSI Design/CAD Symposium; 2000. p. 401–404.
- [10] Bernardi P, Grosso M, Reorda MS, Zhang Y. A programmable BIST for DRAM testing and diagnosis. In: Proceedings of the IEEE International Test Conference. ITC'10; 2010. p. 1–10. doi:10.1109/TEST.2010.5699247.
- [11] Sfikas Y, Tsiatouhas Y. Testing Neighbouring Cell Leakage and Transition Induced Faults in DRAMs. IEEE Transactions on Computers. 2016;65(7):2339–2345.
- [12] Cascaval P, Bennett S, Huțanu C. Efficient March Tests for a Reduced 3-Coupling and 4-Coupling Faults in Random-Access Memories. Journal of Electronic Testing. 2004;20(3):227–243. Available from: <http://dx.doi.org/10.1023/B:JETT.0000029457.21312.23>. doi:10.1023/B:JETT.0000029457.21312.23.
- [13] Huzum C, Cascaval P. A Multibackground March Test for Static Neighborhood Pattern-Sensitive Faults in Random-Access Memories. Electronics and Electrical Engineering. 2012;119(3):81–86.
- [14] Wunderlich HJ. Multiple distributions for biased random test patterns. In: Proceedings of the IEEE International Conference on Test: new frontiers in testing. ITC; 1988. p. 236–244.
- [15] Du X, Mukherjee N, Cheng WT, Reddy SM. Full-speed field-programmable memory BIST architecture. In: Proceedings of the IEEE International Conference on Test. ITC; 2005. p. 1173. doi:10.1109/TEST.2005.1584084.
- [16] Aswin AM, Ganesh SS. Implementation and Validation of Memory Built in Self Test (MBIST) – Survey (September 11, 2019). International Journal of Mechanical Engineering and Technology. 2019;10(3):153–160.
- [17] Harutyunyan G, Shoukourian S, Zorian Y. Fault Awareness for Memory BIST Architecture Shaped by Multidimensional Prediction Mechanism. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2019 March;38(3):562–575. doi:10.1109/TCAD.2018.2818688.
- [18] van de Goor AJ, Offerman A, Schanstra I. Towards a Uniform Notation for Memory Tests. In: Proceedings of the European Design and Test Conference. ED&TC. Paris; 1996. p. 420–427. doi:10.1109/EDTC.1996.494335.
- [19] Yarmolik VN, Klimets Y, Demidenko S. March PS(23N) Test for DRAM Pattern-Sensitive Faults. In: Proceedings of the 7th Asian Test Symposium. ATS'98. IEEE Computer Society; 1998. p. 354–357.
- [20] Nicolaidis M. Transparent BIST for RAMs. In: Proceedings IEEE International Test Conference 1992, Discover the New World of Test and Design, Baltimore, Maryland, USA, September 20-24, 1992. IEEE Computer Society; 1992. p. 598–607.
- [21] Karpovsky MG, van de Goor AJ, Yarmolik VN. Pseudo-exhaustive word-oriented DRAM testing. In: Proceedings of the European Conference on Design and Test. EDTC '95. IEEE Computer Society; 1995. p. 126.

- [22] Kuhn RD, Okum V. Pseudo-Exhaustive Testing for Software. In: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop. IEEE Computer Society; 2006. p. 153–158.
- [23] Wagner KD, Chin CK, McCluskey EJ. Pseudorandom Testing. *IEEE Transactions on Computers*. 1987 Mar;36(3):332–343. doi:10.1109/TC.1987.1676905.
- [24] Das D, Karpovsky MG. Exhaustive and Near-Exhaustive Memory Testing Techniques and their BIST Implementations. *Journal of Electronic Testing*. 1997;10(3):215–229.
- [25] Mrozek I, Yarmolik VN. Iterative Antirandom Testing. *Journal of Electronic Testing*. 2012 Jun;28(3):301–315.
- [26] Yarmolik SV, Zankovich AP, Ivanyuk AA. *Marshevyte testy dlya samotestirovaniya OZU (March Tests for RAM Self-testing)*. Minsk: Beloruss. Gos. Univ.; 2009.
- [27] van de Goor AJ, Kukner H, Hamdioui S. Optimizing memory BIST Address Generator implementations. In: Proceedings of the 6th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS); 2011. p. 1–6.
- [28] Kumar S, Rajkumar M. Efficient Memory Built in Self-Test Address Generator Implementation. *International Journal of Applied Engineering Research*. 2015;10(7):16797–16813.
- [29] Cascaval P, Silion R, Cascaval D. A BIST Logic Design for MarchS(3)C Memory Test BIST Implementation. *Romanian Journal of Information Science and Technology*. 2009;12(4):440–454.
- [30] van de Goor AJ, Jung C, Hamdioui S, Gaydadjiev G. Low-cost, customized and flexible SRAM MBIST engine. In: Proceedings of the 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems; 2010. p. 382–387. doi:10.1109/DDECS.2010.5491749.
- [31] Mukherjee N, Pogiel A, Rajski J, Tyszer J. High Volume Diagnosis in Memory BIST Based on Compressed Failure Data. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2010 March;29(3):441–453. doi:10.1109/TCAD.2010.2041852.
- [32] Saravanan S, Hailu M, Gouse GM, Lavanya M, Vijaysai R. Design and Analysis of Low-Transition Address Generator. In: Zimale FA, Enku Nigussie T, Fanta SW, editors. *Advances of Science and Technology*. Cham: Springer International Publishing; 2019. p. 239–247.
- [33] Sosnowski J. Analyzing BIST robustness. In: Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems; 2001. p. 104–109. doi:10.1109/DFTVS.2001.966758.
- [34] Pavani P, Anitha G, Bhavana J, Raj JP. A Novel Architecture Design of Address Generators for BIST Algorithms. *International Journal of Scientific & Engineering Research*. 2016 Feb;7(2):1484–1488.
- [35] Singh B, Narang SB, Khosla A. Address Counter / Generators for Low Power Memory BIST. *International Journal of Computer Science*. 2011 July;8(1):561–565.
- [36] Yarmolik VN, Yarmolik SV. Generating modified Sobol sequences for multiple run march memory tests. *Automatic Control and Computer Sciences*. 2013;47(5):242–247. Available from: <http://dx.doi.org/10.3103/S0146411613050088>. doi:10.3103/S0146411613050088.
- [37] Savage C. A Survey of Combinatorial Gray Codes. *SIAM Review*. 1996;39(2):605–629.
- [38] Yarmolik VN, Yarmolik SV. The repeated nondestructive march tests with variable address sequences. *Automation and Remote Control*. 2007;68(4):688–698.

- [39] Yarmolik SV, Yarmolik VN. Modified Gray And Counter Sequences For Memory Test Address Generation. In: Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006. Gdynia, Poland; 2006. p. 572–576.
- [40] Chen TY, Merkel RG. Quasi-Random Testing. *IEEE Transactions on Reliability*. 2007;56(3):562–568.
- [41] Yarmolik VN, Sokol B, Yarmolik SV. Counter Sequences for Memory Test Address Generation. In: Proceedings of the 12th International Conference Mixed Design of Integrated Circuits and Systems. MIXDES'05. Krakow, Poland: IEEE Computer Society; 2005. p. 413–418.
- [42] Mrozek I, Yarmolik VN. Problemy funkcjonalnego testowania pamieci RAM. Bialystok, Poland: Bialystok University of Technology; 2009. ISSN 0867-096X.
- [43] Boyd S, Vandenberghe L. Introduction to Applied Linear Algebra Vectors, Matrices, and Least Squares. Cambridge University Press; 2018. doi:10.1017/9781108583664.
- [44] Kolchin VF. Random Graphs. Cambridge University Press; 1998. doi:10.1017/CBO9780511721342.
- [45] Yarmolik VN, Yarmolik SV. Address sequences for multi run RAM testing (In Russ.). *Informatics*. 2014; (2):124–136.
- [46] Yarmolik VN, Mrozek I, Yarmolik SV. Controlled method of random test synthesis. *Automatic Control and Computer Sciences*. 2016;49(6):395–403. Available from: <http://dx.doi.org/10.3103/S0146411615060115>. doi:10.3103/S0146411615060115.
- [47] Yarmolik SV, Yarmolik VN. Memory Address Generation for Multiple Run march Tests with Different Average Hamming Distance. In: Proceedings of the IEEE East-West Design and Test Workshop. EWDTW'06. Sochi, Russia; 2006. p. 212–216.
- [48] Mrozek I, Yarmolik VN. Multiple Controlled Random Testing. *Fundamenta Informaticae*. 2016;144(1): 23–43.
- [49] Mrozek I, Yarmolik VN. Two-Run RAM March Testing with Address Decimation. *Journal of Circuits, Systems, and Computers*. 2017;26(2):1750031. Available from: <http://dx.doi.org/10.1142/S0218126617500311>. doi:10.1142/S0218126617500311.
- [50] Liu H, Chen TY. Randomized Quasi-Random Testing. *IEEE Transactions on Computers*. 2016 June;65(6):1896–1909. doi:10.1109/TC.2015.2455981.