Chapter 26
Computing with Emerging Nanotechnologies

M. Altun

Abstract As current CMOS based technologies are approaching their anticipated limits, emerging nanotechnologies start to replace their role in electronic circuits. New computing models have been proposed. This chapter overviews both deterministic and stochastic computing models targeting nano-crossbar switching arrays and emerging low-density circuits. These models are demonstrated with implementations using Boolean and arithmetic logic. Performance parameters of the models such as area, speed, and accuracy, are also evaluated in comparison with those of conventional circuits.

26.1 Introduction

In 1965, Gordon Moore made an influential prediction about CMOS size shrinking, formulated as the Moore Law stating that the number of transistors on a chip doubles every 18–24 months [1]. His prediction has kept its validity for decades. Nowadays this trend has reached a critical point and it is widely accepted that the trend will end in the next decade. Even Gordon accepted that his prediction will lose it validity in near future [2]. At this point, research is shifting to novel forms of nanoarchitectures including nano-crossbar arrays and probabilistic/stochastic circuits and systems [3–5]. Such technologies have apparent advantages over conventional CMOS technologies, such as high performance capacity and easy manufacturability.

Nano-crossbar arrays are regular and dense structures that are generally fabricated by self-assembly as opposed to lithography based conventional and relatively costly CMOS fabrication techniques [6]. Conventional lithographic techniques face severe challenges for emerging nanotechnologies due to their need for directed

---

M. Altun (✉)
Electronics and Communication Engineering Department, Istanbul Technical University,
Ayazağa Campus, 34469 Maslak, Istanbul, Turkey
e-mail: altunmus@itu.edu.tr

H. Ünlü et al. (eds.), *Low-Dimensional and Nanostructured Materials and Devices*, NanoScience and Technology,
DOI 10.1007/978-3-319-25340-4_26

manipulation of molecules which is quite costly in nanoscale. Along with these advantages in terms of circuit size and fabrication, nano-crossbar arrays have drawbacks including reliability issues and CMOS integration problems, standing against commercial production. While reliability of nano-crossbars have been satisfactorily improved using reconfigurable architectures and new defect tolerance techniques [5, 7, 8], discussed later in this chapter, CMOS integration is still a major problem; CMOL is the strongest candidate for this problem [9]. Indeed, if all parts of a computing system can be successfully realized with nano-crossbar arrays then there will be no need for integration, but current state-of-the art has not reached this point, hopefully in the next decade.

The concept of using stochastic computing models is not new; it dates back to a seminal paper by John von Neumann in 1956 [10]. With the advent of a variety of types of emerging nanoscale technologies, the model has found renewed interest [3, 11]. Unlike conventional CMOS that is solely based on deterministic operations, stochastic circuits use probabilities as inputs and outputs. This feature is invaluable to cope with uncertainties seen in emerging nanotechnologies in the form of variability, reliability, and noise problems [4]. Stochastic computing also offers smaller circuit implementations for arithmetic functions using much fewer transistors compared to conventional CMOS circuits. This feature attracts low density obligated technologies such as printed/flexible electronics [12]. Here, the main drawback is high error rates seen in stochastic computing by nature. Methods, as discussed later in this chapter, have been proposed to improve it.

In this chapter, we focus on computing models for nano-crossbar arrays and stochastic circuits. These models are demonstrated with implementations using Boolean and arithmetic logic. Performance parameters of the models such as area, speed, and accuracy, are also evaluated in comparison with those of conventional circuits. This chapter is organized as follows. In Sect. 26.2, we investigate nano-crossbar array based computing models. We present Boolean function implementation and defect tolerance techniques in Sects. 26.2.1 and 26.2.2, respectively. We evaluate the techniques on standard benchmark circuits in Sect. 26.2.3. In Sect. 26.3, we introduce stochastic computing models and their application areas. We present techniques to reduce error rates and to achieve error free stochastic computing in Sects. 26.3.1 and 26.3.2, respectively. In Sect. 26.4, we present conclusions.

26.2 Computing with Nano-crossbar Arrays

Unlike conventional CMOS that can be patterned in complex ways with lithography, self-assembled nanoscale systems generally consist of regular structures. Logical functions and memory elements are achieved with arrays of crossbar-type switches. In this study, we target this type of switching arrays where each crosspoint behaves as a switch, either two-terminal or four-terminal. This is illustrated in Fig. 26.1. Depending on the used technology, a two-terminal switch based

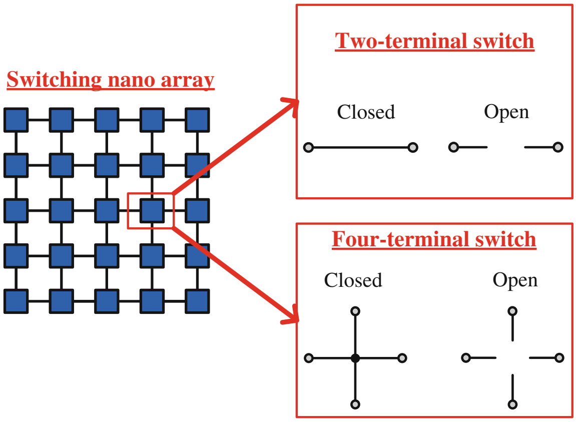

Fig. 26.1 A switching crossbar nanoarray modeled with two-terminal and four-terminal switches

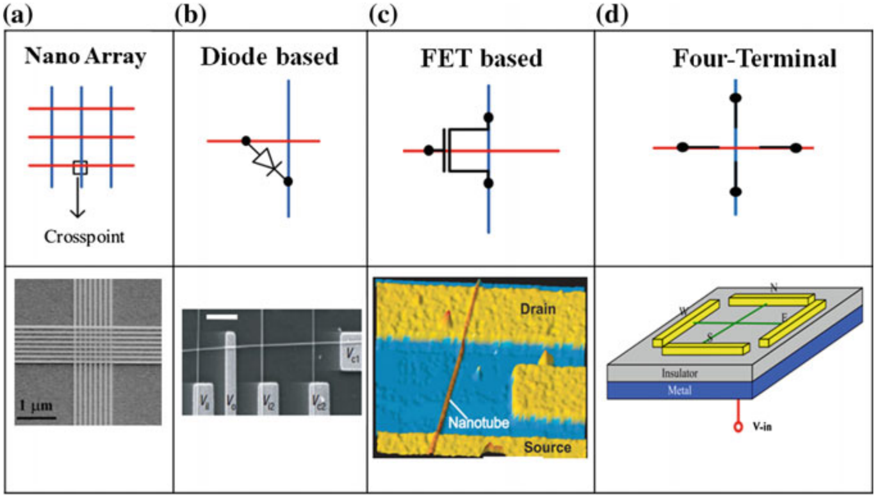

Fig. 26.2 **a** Nano-crossbar [14] based on **b** diode [14], **c** FET [15], **d** four-terminal switch [23] crosspoints

crosspoint can be modeled as a diode [13, 14] or a FET [15, 16]. This is illustrated in Fig. 26.2. Note that both diode and FET based crosspoints conduct current in one direction. However, four-terminal switches conduct current in multiple directions.

We implement Boolean functions by considering array sizes. Table 26.1 compares different implementation methodologies for few XOR functions (Parity functions) regarding array sizes. The columns "diode based" and "FET based" represent two-terminal switch based implementation methodologies. These methodologies have been proposed to implement simple logic functions [17, 18].

Table 26.1 Array sizes for nanoarray computing models; XOR2 = $x_1 \oplus x_2$, XOR3 = $x_1 \oplus x_2 \oplus x_3$, and XOR4 = $x_1 \oplus x_2 \oplus x_3 \oplus x_4$

	Two-terminal switch based nanoarray models		Four-terminal switch based nanoarray models	
	Diode based (Optimal) [13]	FET based (Optimal) [16]	Four-terminal based [23]	Four-terminal based (Optimal) [36]
XOR2	2 × 5 array 10 switches	4 × 4 array 16 switches	2 × 2 array 4 switches	2 × 2 array 4 switches
XOR3	4 × 7 array 28 switches	6 × 8 array 48 switches	4 × 4 array 16 switches	3 × 3 array 9 switches
XOR4	8 × 9 array 72 switches	8 × 16 array 128 switches	8 × 8 array 64 switches	3 × 5 array 15 switches

In this study, we generalize them to be applicable for any given Boolean function with offering optimal array size formulations. The last two columns represent four-terminal switch based implementation methodologies that offer favorably better results.

Defect rates are much higher for nano-crossbars compared to conventional CMOS circuits [19]. Therefore developing new defect tolerance techniques for nano-crossbars is a must, especially for high defect rates up to 20 %. Tolerating such high defect rates necessitates using reconfigurable crossbar architectures and redundancy [5, 20]. A predetermined design with static crossbars is not capable for defect tolerance because it is not possible to create alternative routes for defective regions. On the contrary, reconfigurable designs can be manipulated for defect tolerance. In this study, we assess and compare defect tolerance performances of the three different reconfigurable nano-crossbar architectures/technologies, represented in Fig. 26.2, that is conducted through finding a valid mapping in accordance with the proposed algorithm and defect maps in case of randomly distributed defects. We consider randomly occurred stuck-open and stuck-closed crosspoint defects causing permanently open and closed defective crosspoint devices or switches, respectively.

This study is at the technology-independent level. The presented synthesis and optimization methods are applicable to variety of nanoarray based emerging technologies including nanowire and nanotube crossbar arrays [13, 15–18], magnetic switch-based structures [8], arrays of single-electron transistors [21], and memristive arrays [22].

26.2.1 Implementing Boolean Logic Functions

We investigate three major implementation methodologies developed for switching nanoarrays. We classify them as two-terminal or four-terminal switch based.

26.2.1.1 Two-Terminal Switch Based Methodologies

These methodologies consider each crosspoint of an array as a two-terminal switch that behaves like a diode or a FET. Since diodes and FETs conduct current through their two terminals that are anode and cathode for diodes and source and drain for FETs, they are fundamentally two-terminal switches.

Boolean functions are implemented by using conventional techniques from diode-resistor logic and CMOS logic with an important constraint regarding nanoarray structures. Boolean functions should be implemented in their sum-of-products (SOP) forms; other forms such as factored or BDD (Binary Decision Diagram) cannot be used since these forms require manipulation/wiring of switches that is not applicable for self-assembled nanoarrays. Figure 26.3 shows implementations of a Boolean function XOR2 with diode and with FET based nanoarrays.

Array size formulations: Given a target Boolean function *f*, we derive formulas of the array sizes required to implement *f*. This is shown in Table 26.2. For diode based implementations, each product of *f* requires a row (horizontal line), and each literal of *f* requires a column (vertical line) in an array. Additionally, one extra column is needed to obtain the output. For FET based implementations, each product of *f* and its dual, f^D, requires a column, and each literal of *f* requires a row in

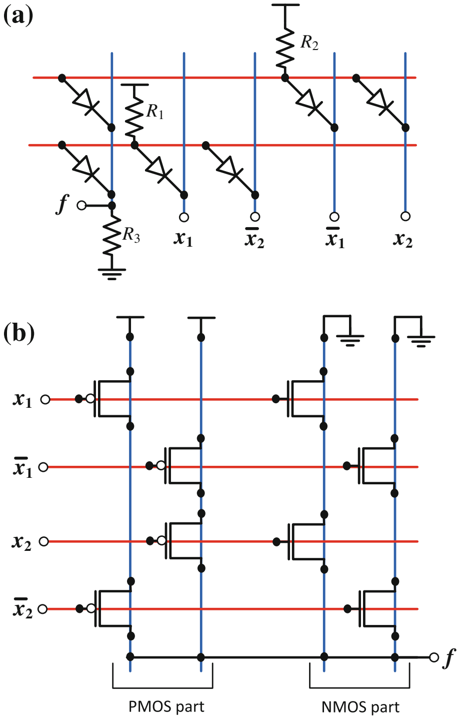

Fig. 26.3 **a** Diode and **b** FET based nanoarrays implementing XOR2 = $x_1 \oplus x_2$ with 2 × 5 and 4 × 4 arrays, respectively

an array. As an example shown in Fig. 26.3: $f = \text{XOR2} = x_1\overline{x_2} + \overline{x_1}x_2$ has 4 literals and 2 products; $f^D = x_1x_2 + \overline{x_1}\,\overline{x_2}$ has 2 products. This results in array sizes of 2 × 5 and 4 × 4 for diode and FET based implementations, respectively. Note that both formulas, for diode and FET, always result in optimal array sizes; no further reduction is possible.

26.2.1.2 Four-Terminal Switch Based Methodology

This methodology considers each crosspoint of an array as a four-terminal switch. An example is shown in Fig. 26.4. The four terminals of the switch are all either mutually connected (ON) or disconnected (OFF). Boolean functions are implemented with top-to-bottom paths in an array by taking the sum (OR) of the product (AND) of literals along each path. This makes Boolean functions implemented in their sum-of-products (SOP) forms. Figure 26.5a, b show the implementations of a Boolean function XOR2 in an array and lattice representations, respectively. Figure 26.5c shows a lattice of four-terminal switches implementing a Boolean function $x_1x_2x_3 + x_1x_2x_5x_6 + x_2x_3x_4x_5 + x_4x_5x_6$. The function is computed by taking the sum of the products of the literals along each path. These products are $x_1x_2x_3$, $x_1x_2x_5x_6$, $x_2x_3x_4x_5$, and $x_4x_5x_6$.

Array size formulation: Given a target Boolean function f, the array size formula was proposed by Altun and Riedel [23] that is shown in Table 26.3. In their implementation, each product of f and its dual, f^D, require a column and a row, respectively, in an array. As an example shown in Fig. 26.5a, $f = \text{XOR2} = x_1\overline{x_2} + \overline{x_1}x_2$ and $f^D = x_1x_2 + \overline{x_1}\,\overline{x_2}$ have both 2 products. This results in an array size of 2 × 2.

Examining the array size formulas in Tables 26.2 and 26.3, we see that while the formulas in Table 26.2 always result in optimal sizes, the sizes derived from the formula in Table 26.3, that is, for four-terminal switch based arrays, are not

Table 26.2 Array size formulas for diode and FET based implementations

Type	Array size formulas (Optimal)
Diode	(Number of products in ***f***) x ("number of literals in ***f***" + 1)
FET	(Number of literals in ***f***) x ("number of products in ***f***" + "number of products in f^D")

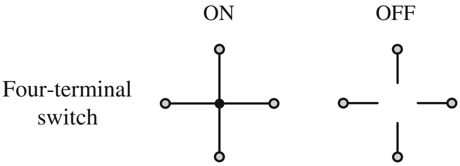

Fig. 26.4 A four-terminal switch with its two states

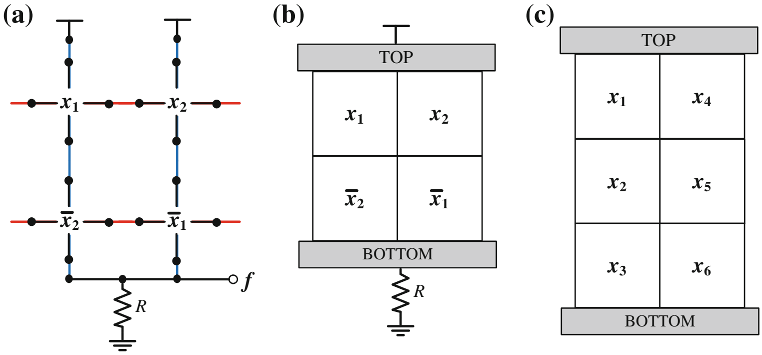

Fig. 26.5 **a** Four-terminal switch based nanoarray and **b** its lattice representation implementing XOR2 = $x_1 \oplus x_2$ with a size of 2 × 2. **c** Four-terminal switch based lattice implementing $x_1x_2x_3 + x_1x_2x_5x_6 + x_2x_3x_4x_5 + x_4x_5x_6$

Table 26.3 Array size formula for four-terminal switch based implementation

Type	Array size formula (Non-optimal)
Four-terminal	(Number of products in f) x (number of products in f^D)

necessarily optimal. In the following part we present an algorithm that finds an optimal size implementation of any given target Boolean function.

Finding whether a certain array with assigned literals to its switches implements a target function is the main problem in finding optimal sizes. This problem requires to check if each assignment of 0's and 1's to the switches, corresponding to a row of the target function's truth table, results in logic 1 (a top-to-bottom path of 1's exists). To check this we have to enumerate all top-to-bottom paths; the size of this task grows exponentially with the array size. This is a general statement that holds also for our algorithm described below.

Our algorithm finds optimal array sizes to implement given target Boolean functions with arrays of four-terminal switches in four steps:

(1) Obtain irredundant sum-of-products (ISOP) expressions of a given-target function f_T and its dual f_T^D. Determine the upper bound on the array size using the formula in Table 26.3.
Upper Bound (UB) = (number of products in f_T) × (number of products in f_T^D). The implementable **lower bound (LB)** values are taken from the lower bound table proposed in [23].

(2) List the array shapes (RxC) (which are in between **LB** and **UB**) into the 'List of Implementable Nanoarray Shapes' and sort them regarding of array sizes, in ascending order. While ordering, first take the array shape which has lower number of rows (e.g. if the kth shape is "3 × 4", then the (k + 1)th shape can be "4 × 3".). Suppose that there are total of **N** different shapes in the list. For step 3, start with n = 1 (1 ≤ n ≤ N).

(3) Compute the value of the following statement for the nth shape.
The Statement: An array which has the shape in the nth line of the list is implementable for f_T.
If the statement is **TRUE**
 Change **UB** to the **R×C** (save the design);
 Go to the step 4;
If the statement is **FALSE**
 Increase the number "n" by 1 (n = n+1);
 Repeat step 3.
(4) Declare that **UB** is **optimal size** for given-target function f_T can be realized in.

Our algorithm is mainly based on finding a design in a certain sized array such that the design implements f_T. Our algorithm does not check every possible design. If it did then it would be intractable even for small sized arrays. For example, if a target function f_T having 6 variables, 8 literals, is tested on a 3 × 4 array then there are 12^{10} possible designs and 2^6 truth table rows. Note that for each of the 12 switches in the array there are 10 different options; it might be one of the 8 literals, 0, or 1. In this scenario, the algorithm would have to check $12^{10} \times 2^6$ truth table rows. To overcome this problem, we discard a significant portion of designs to be checked. For this purpose, we offer 3 major improvements:

I. We create a library of reduced number of R × 2 sized sub-designs. We use them to achieve R × C sized designs. While creating sub-designs we exploit the following simple lemmas. First lemma allows us to discard designs implementing a product (s) that does not imply f_T. The second lemma allows us to discard designs with "0" assignments to the switches if f_T has a product having a single literal.

Lemma 1 *If a design has a path realizing a product p for which* $f_T \neq f_T + p$, *then the design can not implement* f_T.

Proof Since p is not an implicant of f_T, then a design including p implements a different function.

Lemma 2 *If a function* f_T *has a single variable product term* $p = x$ *then the algorithm does not need to assign "0" to the switches.*

Proof All the "0" assignments can be replaced with x's without a loss of generality.

II. If there is a product of f_T such that the number of literals of the product equals to the number of switches in the longest top-to-bottom path in the array, then we settle that particular product onto that particular path.
III. We discard designs having fewer number of total literals than the total number literals of f_T.

These improvements make our algorithm much faster. As an example, suppose that XOR3 is given as a target function for which the improved algorithm runs roughly 400 times faster. For 3 × 2 sized sub-designs, there are $8^6 = 262{,}144$ designs.

With applying the proposed improvements, this number is reduced to 12,114, roughly 20 times smaller than the unimproved one. Since we use two sub-arrays for XOR3, for the optimal array size of 3 × 4, the improved algorithm works 400 times faster.

26.2.2 Defect Tolerance

In this section, defect tolerance performance of switching nano-crossbar arrays is extensively studied. Three types of nanoarrays where each crosspoint behaves as a diode, FET, and four-terminal switch, are considered. For each crosspoint, both stuck-open and stuck-closed defect probabilities are independently taken into consideration. A fast heuristic algorithm using indexing and mapping techniques is presented [20]. The algorithm measures defect tolerance performances of the crossbar arrays that are expected to implement a certain given function. The algorithm's effectiveness is demonstrated on standard benchmark circuits that shows 99 % accuracy compared with an exhaustive search. The benchmark results, presented in the next section, also show that not only the used technology, the nanoarray type, but more significantly the specifics of given functions affect defect tolerance performances.

Mapping a target logic function on a defective crossbar is an NP-complete problem [24]. In the worst-case scenario, an N × M crossbar has N!M! permutations that is intractable for a reasonable computing time. Different algorithms and heuristics are presented to tackle this issue. Graph based models are proposed in [25, 26] that use a fan-out embedding heuristic and a maximum flow algorithm, respectively. In addition to graph based approaches, "Integer Linear Programming" is used in [27] that employs a pruning centered approach with certain constraints. It is shown in [27] that defect tolerance results might dramatically depend on the chosen algorithm correlated to the algorithm's accuracy. We test and compare our algorithm with an exhaustive search to establish an accuracy of 99 %. However, it should be noted that large crossbars are computationally intractable to be included in exhaustive search so we only consider crossbars up to 7 × 7 size for comparison. In addition, the approaches mentioned so far are using pre-determined crossbar sizes to find a mapping for a chosen logic function. We use an optimal crossbar to realize a logic function which means that both the function and the crossbar matrices have the same size. We test logic functions in Irredundant Sum-of-Products (ISOP) form that is consistent with using optimal crossbar sizes. Also we include three different logic families for comparison which departs from the mentioned studies in the literature.

We present a heuristic algorithm that creates and compares index representations of a given function and a defective certain sized crossbar to be used to implement the function. We show that if the index representations are not matched then the defective crossbar cannot be used to implement the function; otherwise, it can be used. We prove that this is a necessary and sufficient condition. Our algorithm eliminates considerable amount of crossbar mapping permutations that is the main headache for the mentioned studies in the literature. Furthermore, our viewpoint is comprehensive, that is, concerned both with the types of the crossbar technologies and the characteristics of given functions. The presented indexing based algorithm is direct used for diode and CMOS based logic of nano-crossbars. For four-terminal switch based logic, the presented algorithm is partially used; a conventional matrix-based matching is mainly performed. Also we include three different logic families for comparison which departs from stated studies. It is important to determine features of different families due to post-production selection according to inherent defect types.

As follows, we first explain the algorithm used for diode and CMOS based logic of nano-crossbars [20]. Then, we briefly explain the defect tolerance technique used for four-terminal switch based logic. In the next section, we present experimental results and elaborate on them.

26.2.2.1 The Algorithm for Diode and CMOS Based Logic

The outline of our four-step algorithm is shown below. We will then explain each step in details. The algorithm will be demonstrated with an example in Fig. 26.6. It should be noted that the example and the following explanations are for stuck-open defects, nevertheless they can be applied easily for stuck-closed defects by considering defects as 1s (as opposed to 0s) to be matched with 1s (as opposed to 0s) in the function matrix.

Input: Function matrix and crossbar (defective) matrix
Output: If there is a matching, “YES”; otherwise “NO”

Step 1: If the number of defective switches is greater than the corresponding elements in the function matrix, then return “NO”.
Step 2: Sort matrices according to the row and column index, if the crossbar matrix has at least one row or column index greater than a row or column index of the function matrix, return “NO”.
Step 3: If the number of defects is equal to or smaller than the worst-case limit W_C, return “YES”.
Step 4: Find the reduced matrix and find set of double indices. Start subarray search. If a subarray is found with the equal set of double indices as the reduced matrix with 20,000 trials, return “YES”; otherwise return “NO”.

The explanations of the algorithm steps for stuck-open defects:

Step 1: *If the number of defective switches is greater than the corresponding elements in the function matrix, then return "NO".*
We consider stuck-open defects so algorithm checks 0s in the crossbar matrix to match 0s in the function matrix. If 0s in the crossbar matrix is greater than 0s in the function matrix, then it is not possible to find mapping.

Step 2: *Sort matrices according to the row and column index, if the crossbar matrix has at least one row or column index greater than a row or column index of the function matrix, return "NO".*
We sort matrices according to the row and column indices. For example, in Fig. 26.6 index sets of the sorted function and crossbar matrices for 0 are:

$$\mathrm{I}_{\mathrm{R,F}} = \{2,\ 2,\ 2,\ 1\}\ \mathrm{I}_{\mathrm{C,F}} = \{2,\ 2,\ 2,\ 1\}$$
$$\mathrm{I}_{\mathrm{R,C}} = \{2,\ 1,\ 0,\ 0\}\ \mathrm{I}_{\mathrm{C,C}} = \{2,\ 1,\ 0,\ 0\}.$$

There is a perfect matching between sets. However, if a member of a crossbar set would be greater than corresponding member in a function set, there would be no matching. This would mean there are excessive

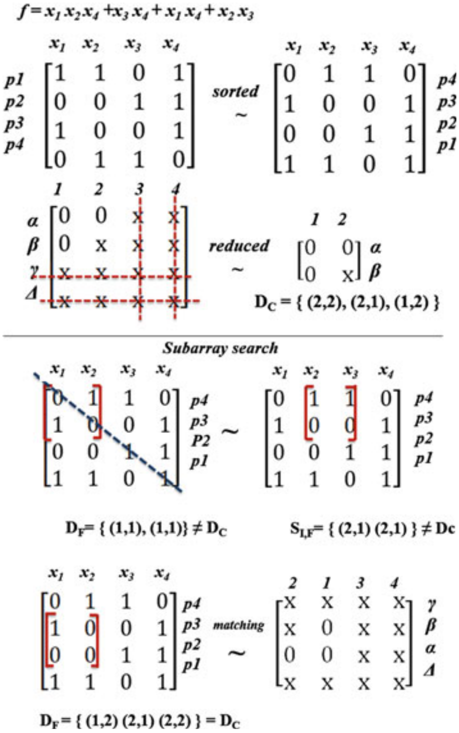
f = x1x2x4 + x3x4 + x1x4 + x2x3
x1 x2 x3 x4
p1 p2 p3 p4
sorted
p4 p3 p2 p1
1 2 3 4
α β γ Δ
reduced
Dc = { (2,2), (2,1), (1,2) }
Subarray search
DF = { (1,1), (1,1)} ≠ DC
SI,F = { (2,1) (2,1) } ≠ Dc
matching
DF = { (1,2) (2,1) (2,2) } = DC

Fig. 26.6 Fourth step of the presented algorithm: sorting the function matrix, crossbar reducing, and subarray search

defective elements for matching. For the example in Fig. 26.6, this is not an issue so we proceed to the step 3.

Step 3: *If the number of defects is equal to or smaller than the worst-case limit* W_C*, return "YES".*

The worst-case limit of a function matrix is the maximum number of tolerable defects in any defect distribution related to the row and column index. We find W_C with using sets of row and column indices. First we choose minimum members in sets of row and column indices, separately. After that we choose the minimum between two members obtained in the first step. W_C gives us minimum row and column index. If crossbar matrix has defective elements less than or equal to Wc, then defects can be matched with any row and column in a function matrix. For illustration, let us now follow the process of finding the Wc of the function matrix shown in Fig. 26.6.

$$I_{R,F} = \{2, 2, 2, 1\} I_{C,F} = \{2, 2, 2, 1\}$$

$\min\{I_{R,\ F}\} = 1$ and $\min\{I_{C,\ F}\} = 1$ so $W_C = 1$

In Fig. 26.7, the crossbar matrix has three defective elements, so we cannot conclude if there is a matching without performing the fourth step.

Step 4: *Find the reduced matrix and find set of double indices. Start subarray search. If a subarray is found with the equal set of double indices as the reduced matrix with 20,000 trials, return "YES"; otherwise return "NO".*

In a crossbar matrix, Xs corresponding to functional switches do not change the double index of a matrix element. For this reason, we erase the columns and rows consisting of only such elements (Xs) for compactness. The acquired matrix keeps the same set of double indices. Figure 26.7 shows an example for this.

In the next step, we use a subarray search to find a matching between a reduced matrix and a subarray. Then, the function and crossbar matrices are sorted according to the sets of indices. We use this to increase the chance of finding matrices with the same set of double indices. Since matching elements are collected to the one side, the search progresses diagonally. It can be seen from Fig. 26.6 that searching a subarray checks only the set of double indices of a chosen subarray, due to the Double Index Theorem, presented below. As long as they have the same set of double indices, permutation of matrices is not necessary. Once two matrices are found with the same set, it means there is a mapping, so the algorithm returns "YES".

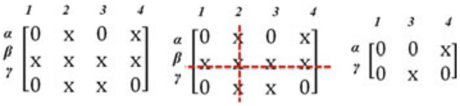

Fig. 26.7 Reduced matrix for subarray search

Double Index Theorem *There is a one-to-one matching between two matrices if and only if their set of double indices are equivavelent.*

Lemma 1 *Row and column permutations do not alter the double index of a matrix element.*

Numbers of the double index are defined with the row and column indices in which the element is found. Therefore even though permutations changes the position of a row or a column, element is still in the same row and column with the same row and column indices which defines the double index.

Lemma 2 *Set of double indices is unique for a given matrix.*

Proof The proof is by contradiction. Let's assume such a matrix that has two different set of double indices. Therefore sets should have different double indices for one element or more. Double indices in a matrix are determined according to a row and column index. Since matrix is not altered and has same number of elements in rows and columns for a chosen value of 0 or 1, it is not possible to have different row and column indices that comprises double indices. This contradicts with the assumption of having different double index for an element or more.

Lemma 3 *Set of double indices for a matrix does not change with row and column permutations.*

Proof Rows and columns of a matrix is consisted of matrix elements which have the same double index after permutations according to Lemma 1. Therefore set stays the same since its members are not changed.

Proof of the Double Index Theorem Sufficiency If there is a one-to-one matching between two matrices, then they are identical by definition. Therefore their set of double indices is equivavelent according to Lemma 2.

Necessity Lemma 2 states that set of indices is unique for a certain matrix and Lemma 3 states that set of double indices stays the same after row and column permutation. Therefore, if two matrices have the same set of double indices, then they are either identical or permutation of one another; in both case they can be matched one-to-one.

Subarray search has a trial limit of 20,000. We choose this value because when compared with exhaustive search, it gives 99 % accuracy. When we run the algorithm by removing this limitation, we see that there is almost no change in the values. This validates the heuristic algorithm presented above interpreted as a negative result meaning that there is no mapping at all. In subarray search, the double index theorem is only valid for matrices with the same number of elements to be matched. In case of unequal number of elements, new defective elements should be introduced to the reduced matrix to equalize the number of elements. If there are N missing elements for matching, 2^N possibilities are to be considered. Instead of doing this, functional switches denoted with x are marked with 0 and defective switches with 1. Same is applied to the subarray in the function matrix. Next, element by element multiplication of matrices is executed. Since functional switches can be matched with 0s and 1s in the function matrix, the resulting matrix can be compared with the reduced matrix. If they are equal, there is a matching.

In the CMOS based design two matrices are used to model a logic function. All principles used for diode based design are valid, with the exception of input permutations. In the diode based design both inputs (columns) and products (rows) can be permutated with respect to the crossbar which are checked for a mapping. In the CMOS based design, the first matrix is for the function itself and the second matrix for its complement. Important distinction is that inputs are in the same order for both matrices. For this reason while a mapping is searched for a crossbar, rows of matrices can be permutated independently; however inputs must be in the same order for both matrices.

26.2.2.2 Defect Tolerance for Four-Terminal Switch Based Logic

For four-terminal switch based design, we partially use the above presented algorithm. We use matrix based defect maps similar to those used there. Our defect tolerance technique mainly depends on permutation trials. Therefore, we mention it only briefly.

Defect Map Due to its layout method, the four-terminal switch based design [17] uses every switch on the crossbar. Therefore there is no unused switch like those in diode and CMOS based designs. However, certain functions yield redundant paths or extra connections between top and bottom plates. Figure 26.8 shows occurrence of extra connections. If a defect existing in a crossbar appears only on one of the connections, then it can be compensated with the other connection and the correct result can still be achieved. A defect map of a function implemented with four-terminal switch based design displays these type of connections.

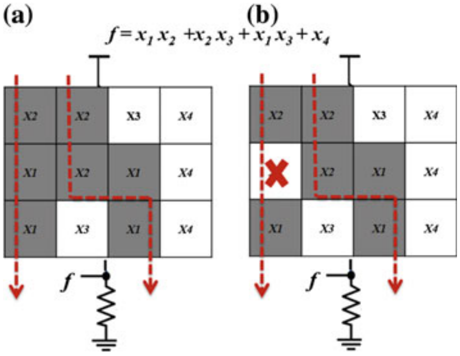

Fig. 26.8 **a** Four-terminal implementation of f with two connections between *top* and *bottom* plates; **b** in case of a stuck-open defect, first connection is broken down, however since there is a second connection f evaluates correctly

26.2.3 Simulation Results

In Table 26.4, we report synthesis results for standard benchmark circuits [28]. We treat each output of a benchmark circuit as a separate target function. The number of products for each target function f_T and its dual f_T^D are obtained through sum-of-products minimization using the program Espresso [29]. The array size values for “Diode”, “CMOS”, and “4-terminal” are calculated by using the formulas in Tables 26.2 and 26.3. The array size values for “Optimal 4-terminal” are obtained using the presented optimization algorithm in Sect. 26.2.1: Implementing Boolean logic functions.

Examining the numbers in Table 26.4, we always see the same sequence from the worst to the best result as “CMOS”, “Diode”, “4-terminal”, and “Optimal 4-terminal”. This proves that models based on four-terminal switches overwhelm those based on two-terminal switches regarding the array size. Further, the numbers obtained by our optimal synthesis method compares very favorably to the numbers obtained by previous methods.

In Table 26.5, we use standard benchmark circuits to measure defect tolerance performances of different nano-crossbar technologies. We consider stuck-open and stuck-closed defect probabilities/rates of 10 and 20 % for each crosspoint independently. Simulations are conducted in Matlab. Crossbars with random defects are produced with Matlab’s predetermined function generator. To obtain defect tolerance values, a sample size of around 600 is used. At this level the defect tolerance fluctuation stabilizes. All experiments run on a 1.70-GHz Intel Core i5 CPU (only single core used) with 4.00 GB memory. It takes 0.2 s for each sample in average to check a valid mapping that satisfies an accuracy of 99 % compared with an exhaustive search.

Table 26.5 shows the results of benchmark functions with respect to defect rates and defect types as well as the crossbar technologies. Considering the technologies and the related logic synthesis methodologies, the diode based logic always has a better defect tolerance performance compared with the CMOS based one. The reason behind this is directly connected to the number of matchings necessitated for valid mapping. Since the CMOS based logic uses two different planes for function realization, it needs to satisfy two matchings instead of one. Another important conclusion is that the four-terminal switch based design yields better results for stuck-closed defects than for stuck-open ones since the design generally requires to assign the same literals to multiple switches on the same conduction path. Characteristics of the functions also play an important role in the defect tolerance. Since stuck-open defects are tolerated with zeros in the functions’ matrices, functions with relatively higher number of products compared to their number of literals have a better chance for tolerating these defects. On the contrary, functions with relatively higher number of literals compared to their number of products have a better chance for tolerating stuck-closed defects. For example, Ex33 in Table 26.1 has a 14 % defect tolerance for stuck-closed and 99 % for stuck-open type defects.

Table 26.4 Array sizes of three different nano-crossbar based logic families

Benchmark	CMOS	Diode	4-terminal	Optimal 4-terminal
Alu 0	30	18	6	6
Alu 1	30	18	6	6
Alu 2	30	18	6	6
Alu J	30	18	6	6
B12 0	80	32	24	12
B12 1	120	70	35	16
B12 3	30	20	8	8
B12 4	42	28	8	3
B12 6	132	77	35	18
B12 7	110	66	24	18
B12 8	90	70	14	14
C17 0	36	18	9	6
C17 1	30	20	8	8
Clpl 0	64	32	16	12
Clpl 1	36	18	9	9
Clpl 2	16	8	4	4
Clpl 3	144	72	36	18
Clpl 4	100	50	25	15
Del 1	25	10	6	6
Del 2	72	36	16	12
Del 5	35	15	12	6
Del 6	36	18	9	6
Ex5 31	156	104	32	24
Ex5 33	110	77	21	21
Ex5 46	81	54	18	18
Ex5 49	72	54	12	12
Ex5 50	81	63	14	14
Ex5 61	64	48	12	12
Ex5 62	49	35	10	10
Misexl 1	48	16	8	8
Misexl 2	132	55	35	15
Misexl 3	156	60	40	24
MLsexl 4	121	44	28	16
Misexl 5	90	45	25	15
Misexl 6	143	66	42	18
Misexl 7	81	36	20	15
Mp2d 4	345	75	90	24
Newrag	108	72	32	18

Table 26.5 Defect tolerance performances of three different nano-crossbar based logic families

Circuit name	Diode				CMOS				Four-terminal			
	Stuck-closed		Stuck-open		Stuck-closed		Stuck-open		Stuck-closed		Stuck-open	
	10 %	20 %	10 %	20 %	10 %	20 %	10 %	20 %	10 %	20 %	10 %	20 %
Alu 0	85	54	99	96	63	50	93	72	86	64	52	26
Alu 1	85	54	99	96	83	53	97	73	86	64	52	26
Alu 2	85	54	99	96	83	53	97	73	86	64	52	26
Alu 3	85	54	99	96	83	53	94	73	86	64	52	26
B12 0	98	80	98	74	46	5	95	59	28	7	28	7
B12 1	92	33	99	75	58	1	99	91	36	9	58	19
B12 3	96	77	96	78	90	58	93	68	42	17	43	16
B12 4	84	40	99	96	79	25	93	74	42	17	43	16
B12 6	68	2	95	47	14	1	99	85	27	7	21	11
B12 7	44	2	99	95	34	1	82	75	42	11	22	3
B12 8	32	1	99	97	24	1	99	99	82	40	22	4
C17 0	95	78	99	94	92	70	98	87	53	26	53	26
CI7 1	96	77	96	78	91	64	92	69	43	16	43	16
ClplO	97	69	99	98	78	23	99	92	62	29	53	20
Clpl 1	98	84	99	95	98	83	98	82	39	14	48	20
Clpl 2	97	82	99	94	93	79	98	92	67	40	65	42
Clpl 3	87	53	99	81	49	1	50	21	18	3	41	10
Clpl 4	91	41	99	97	74	6	63	50	18	3	41	12
Del 1	99	97	95	75	84	52	93	73	52	25	52	25
Del 2	93	55	99	96	68	9	99	96	28	6	28	6
Del 5	99	95	97	85	96	84	84	53	65	38	53	26
Del 6	95	79	99	88	94	70	98	86	53	25	53	25
Ex5 31	56	5	99	95	30	1	83	64	35	7	22	2
Ex5 33	14	1	99	98	9	1	60	43	66	28	25	4
Ex5 46	45	5	99	99	38	1	84	65	17	1	28	6
Ex5 49	3	0	99	99	1	1	84	65	90	32	28	6
ExS 50	23	1	99	99	22	1	87	45	93	75	22	4
Ex5 61	29	2	99	99	25	1	98	78	90	32	43	16
Ex5 62	28	1	98	85	23	1	96	74	95	74	37	12
Misexl 1	99	96	92	66	65	17	52	9	44	16	43	16
Mlsexl 2	78	18	99	92	30	1	98	87	29	6	36	10
Misexl 3	94	38	99	86	10	1	96	67	8	1	38	11
Misexl 4	93	44	99	94	8	1	99	89	27	5	38	10
Misexl 5	86	45	97	80	63	3	95	64	26	4	42	14
Misexl 6	89	28	99	86	26	1	93	73	12	2	29	5
Misexl 7	95	57	99	92	50	1	99	93	21	3	49	15
Newtag	59	4	99	98	52	1	96	52	62	22	30	7

Significant difference between values occurs because of the mentioned literal related properties. Using Ex33 is more favorable for a crossbar with higher probability of stuck-open type defects.

26.3 Stochastic Computing

Stochastic computing (SC), represents values in time domain by random bit streams [29] and was first presented in a paper written by John von Neumann in 1956 [10]. These values are interpreted as probabilities. Therefore, the available range is [0,1] interval. For instance, if a 16 bit length stream contains 12 1s, it represents p = 0.75 independently from the position of 1s in the stream. For example, both (0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1) and (0,1,0,1,1,1,1,1,1,0,1,1,1,1,1,0) represent the same probability p = 0.75. The same probability values can be obtained with different bit stream lengths. Complicated stream operations can be performed with simple circuits in SC. Multiplication and scaled addition are basic arithmetic operations of SC.

Stochastic multiplication can be realized with one AND gate independently of the input bit stream length. In contrary to that, conventional (digital) CMOS multipliers need about 1000 gates for 16 bit multipliers and this number is exponentially increasing with the number of bits. Due to the interval [0,1], addition cannot be performed in SC. Instead, scaled addition that guarantees to remain in the interval is described. MUX is used for scaled addition. Figure 26.9 shows stochastic multiplication and scaled addition operations.

Error rates

Different combination of streams represent same values and this yields error rates in SC. These error rates are extremely high for short bit streams. To reduce error rates, longer bit streams should be used. Figure 26.10 illustrates this; in order have error rates below 1 %, streams having more than 1000 bits are needed. These extremely long streams extend the operation time. Hence, classical SC can not be effectively used in mathematical operations.

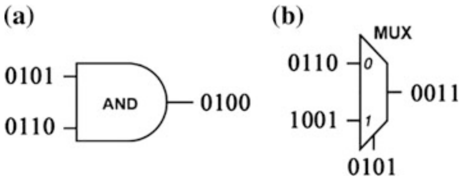

Fig. 26.9 Stochastic implementation of arithmetic operations: **a** Multiplication. **b** Scaled addition

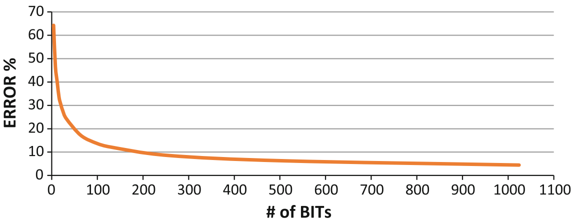

Fig. 26.10 Error rates versus number of bits in the stream for AND gate while $p_1 = 1/2$ and $p_2 = 1/2$

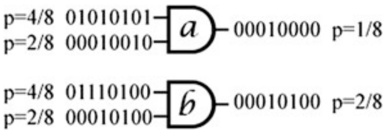

Fig. 26.11 Stochastic multiplication with and without an error

We calculate this error as

$$\frac{\left|z - z_{expected}\right|}{z_{expected}} \tag{26.1}$$

where z is the probability value of the output. And expected value varies for different logic gates.

In order to get “1” at the output of an AND gate, both of the inputs have to be “1”. Assume that the probability of being “1” is p_1 for the 1st input and p_2 for the 2nd input. Hence expected value of multiplication/AND gate is $z = p_1\ p_2$. Following the same logic, expected value at the output of a MUX is $z = p_1\ p_s + p_2\ (1 - p_s)$.

Let two input probability values be p_1 and p_2 respectively depending on the number of 1s in streams. The expected value at the output is $p_1 \times p_2$. However, the same probability obtained with different permutations of input may not always yield the expected results. At this point error rates should be considered. This is illustrated in Fig. 26.11. The expected value can be obtained from the AND gate symbolized with a. However, output value with an error rate is obtained from the AND gate symbolized with b in Fig. 26.11.

Effect of dependency in stochastic computing

In conventional SC, bit streams which have Bernoulli distribution are independent from each other. If input streams become dependent, which is not desire, expected result for classical SC will change. If the inverse of the first input is applied as the second input to an AND gate, “0” will be obtained at the output. It is expected that $z = p\ (1 - p)$, but due to dependency this expectation fails. Moreover, if the same inputs are applied, the output will be same as the input. It acts like a buffer. It is

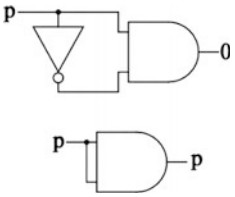

Fig. 26.12 Effect of dependency in SC

expected that z = p p, but due to dependency this expectation fails. These examples are shown in Fig. 26.12.

Application areas

SC needs less area than conventional hardware implementations [4]. On the contrary, error rates are worse. Hence, the commercial application fields are limited (image processing, LDPC codes) [30]. Recently, printed/flexible electronics becomes a major candidate for stochastic computing. Printed electronics strictly requires low density circuits that is a perfect fit with stochastic computing if error rates can be reduced to proper values [31]. This is our motivation. In this study, we offer a method based on improving the generation of bit streams. In Sect. 26.3.1, a detailed information about our method and the conventional one is given. Comparative analysis made between two methods demonstrate our methods success. We also propose a method to achieve error free stochastic computing in Sect. 26.3.2

26.3.1 Reducing Error Rates

We analyze and make comparison between two SC methods, namely random bit assigning and shuffling methods.

26.3.1.1 Random Bit Assigning Method

In order to generate bit streams in SC, conventional random bit generators are used [30]. We name this technique as random bit assigning method (RBAM) [32]. Let desired input probability value for RBAM be p_1 and let random number generator generate values between [0–1] with binomial distribution. If the generated value is smaller than p1, “1” is added to the stream. If the generated value is bigger than p_1 “0” is added to the stream. Because this is a stochastic procedure, there exists difference between the desired and the obtained input values. There are three 1s and five 0s in the generated 8 bit stream in Table 26.6. For this reason, generated input bit stream is physically $p_e = 0.375$. Consequently, the result at the output will be faulty.

Table 26.6 8 bit length stream generated with RBAM for $p_1 = 0.5$

Generated number from generator	Bit stream
0.89967	0
0.45847	1
0.81537	0
0.20818	1
0.71289	0
0.43615	1
0.64214	0

26.3.1.2 Random Bit Shuffling Method

Let the input desired value for random bit shuffling method (RBSM) be p_1. In this method, depending on the desired probability value and the length of the bit stream, a bit stream containing necessary and sufficient amounts of 1 is generated. Fisher-Yates shuffling algorithm [33], adapted to computers by Durstenfeld [34], is applied. In this way, it is guaranteed that the desired and the generated input values are probabilistically identical. With this approach output error rates will be decreased. We use basic logic gates to compare these two methods. The results are obtained by fixing the two inputs, changing the inputs between [0, 1] interval and using different bit lengths.

RBSM–RBAM Comparison for $p_1 = 1/2$, $p_2 = 1/2$ Bit Streams

As shown in Table 26.7, shuffling method gives better results for AND and OR gates. The longer bit length reduces the error rates as expected and the difference of error rates between two methods also is reduced. The expectation is that error rates of two methods overlap at infinity. Two methods give similar results for XOR gate between 4 and 128 bits, however for longer bit streams shuffling method gives better results.

RBSM–RBAM Comparison While 64-Bit Inputs p_1 and p_2 Changing

The graphics in Fig. 26.13 are symmetrical with respect to the diagonal line. Two inputs for the ideal operating range of AND gate should be 0.5 and/or greater. Average error rates for RBAM and RBSM are 30.11 and 19.31 %, respectively. The results are better for RBSM.

The graphics in Fig. 26.14 are symmetrical with respect to the diagonal line. The error rates of an OR gate are better than those for an AND gate. Two inputs for the ideal operating range of an OR gate should be 0.3 and/or greater. Average error rates for RBAM and RBSM are 6.07 and 2.24 %, respectively. The results are better for RBSM.

Table 26.7 Obtained error rates with different bit lengths and three logic gates (%)

Number of bit	RBAM			RBSM		
	AND	OR	XOR	AND	OR	XOR
4	64.3	20.82	37.55	32.55	11.2	33.65
8	46.58	15.38	27.1	26.44	8.77	25.76
12	39.92	12.98	22.52	21.22	7.24	21.85
16	33.38	11.05	19.4	19.04	6.42	19.09
20	30.28	10.22	17.68	17.58	5.85	17.24
24	27.75	9.38	16.35	16.01	5.2	15.66
32	24.28	8.2	14.02	13.82	4.67	13.18
64	17.1	5.68	9.65	8.32	3.05	10.24
100	13.62	4.58	7.88	7.93	2.58	7.82
128	12.12	4.05	7.08	6.21	2.23	7.1
256	8.52	2.8	4.98	3.98	1.6	3.57
512	6.22	2	3.55	2.6	1.34	2.03
1024	4.42	1.4	2.45	1.95	0.35	1.56

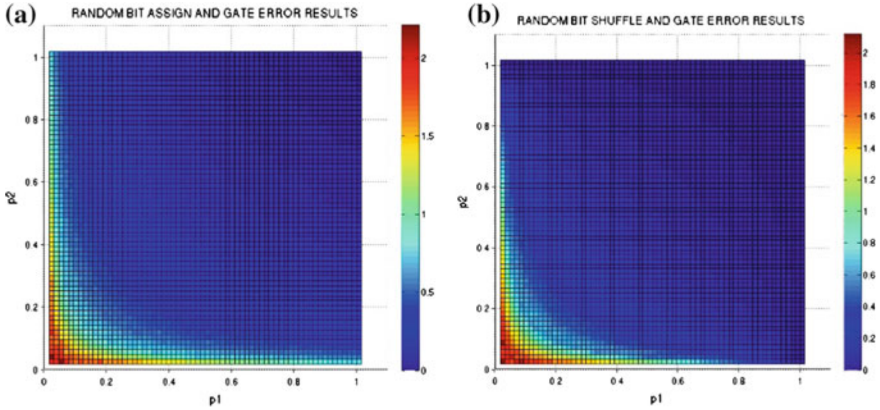

Fig. 26.13 p_1-p_2-error graph for **a** RBAM and **b** RBSM AND gate

The graphics in Fig. 26.15 are symmetrical with respect to both diagonal lines. The error rates of an XOR gate are better than those for an AND gate but worse than those for an OR gate. Two inputs for the ideal operating range of XOR gate should be 0.3 and/or greater. However, the two inputs should not take place on the diagonal line at the same time. Average error rates for RBAM and RBSM are 10.94 and 6.58 %, respectively. The results are better for RBSM.

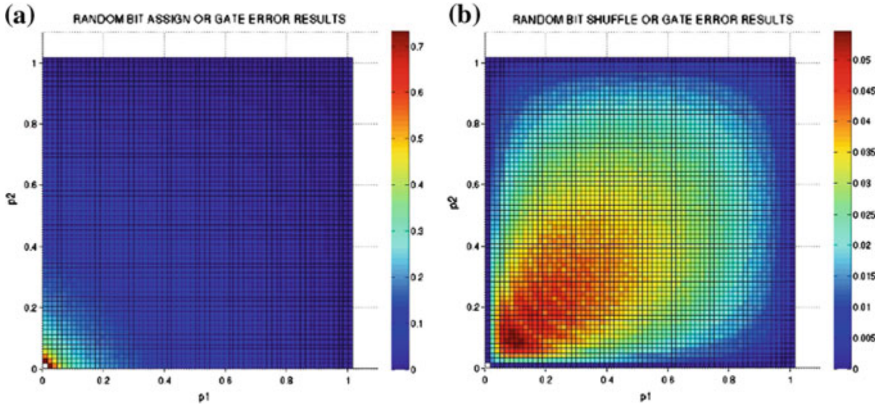

Fig. 26.14 p_1-p_2-error graph for **a** RBAM and **b** RBSM OR gate

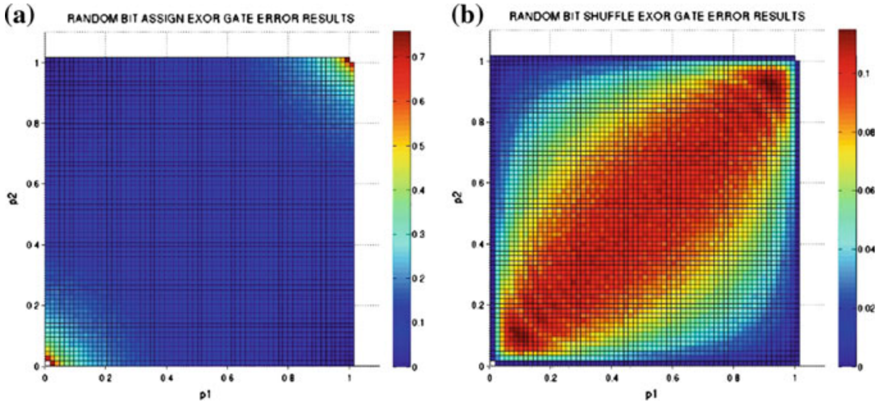

Fig. 26.15 p_1-p_2-error graph for **a** RBAM and **b** RBSM XOR gate

26.3.2 Error Free Stochastic Computing

Conventional wisdom in stochastic computing is that input bit streams should be independent; we previously discussed potential accuracy problems related to dependencies. In this work, however, we use dependent inputs to improve accuracy; we achieve error free outputs.

26.3.2.1 Realizing Error Free Multiplication with 0.5

We suggest an error free block for multiplication by 0.5 shown in Fig. 26.16. With this block, the input stream is multiplied by 0.5 and zero error at the output stream is achieved. However if the length of the input stream is n, the length of the output stream will be 2n because of the merging operation. The merge block can be realized with a delay switch. Thus, the stream obtained from the second AND gate waits the stream from the first AND gate to be finished.

26.3.2.2 Generating Any Probability Value Without an Error

We adapt the block in Fig. 26.16 to switch circuits [35] in order to obtain any probability value. It is illustrated in Fig. 26.17. Inputs are always 0.5 (0.12 = 1/2). In order to shift the binary value one digit to the right and add “0” to the most significant bit, the circuit block based on AND gate is used (0.012 = 1/4). In order

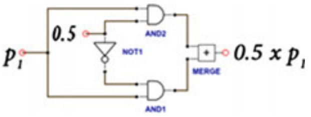

Fig. 26.16 Error free multiplication for 0.5 p_1

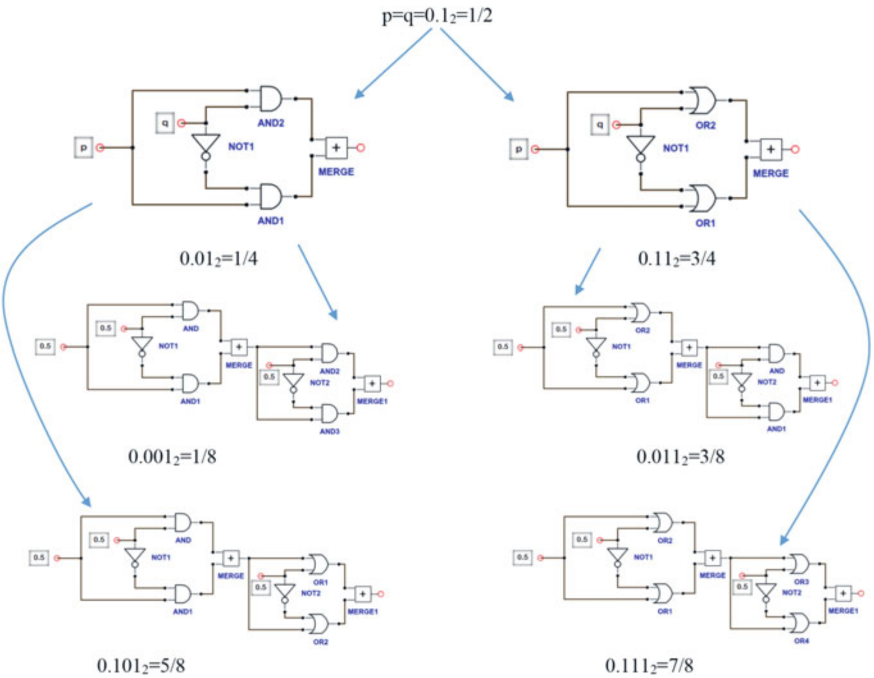

Fig. 26.17 Probability generation tree

to shift the binary value one digit to right and add “1” to the most significant bit, the circuit block based on OR gate is used (0.112 = 3/4). By cascading the blocks, any desired probability value can be accurately obtained without an error.

26.4 Conclusions

This chapter overviews both deterministic and stochastic computing models targeting nano-crossbar switching arrays and emerging low-density circuits. These models are demonstrated with implementations using Boolean and arithmetic logic. Performance parameters of the models such as area, reliability, and accuracy, are also evaluated.

Acknowledgments This work is supported by TUBITAK (The Scientific and Technological Council of Turkey) Career Program #113E760 and EU (European Union) H2020-MCSA Research and Innovation Staff Exchange Program #691178.

References

1. L. Wilson, International technology roadmap for semiconductors (ITRS). Semiconductor Industry Association (2013)
2. M. Dubash, Moore’s Law is dead, says Gordon Moore. Techworld (April 2005)
3. M. Haselman, S. Hauck, The future of integrated circuits: a survey of nanoelectronics. Proc. IEEE **98**(1), 11–38 (2010)
4. W. Qian, X. Li, M.D. Riedel, K. Bazargan, D.J. Lilja, An architecture for fault-tolerant computation with stochastic logic. IEEE Trans. Comput. **60**(1), 93–105 (2011)
5. D.B. Strukov, K.K. Likharev, Reconfigurable nano-crossbar architectures. Nanoelectronics Inf. Technol. 543–562 (2012)
6. G.M. Whitesides, B. Grzybowski, Self-assembly at all scales. Science **295**(5564), 2418–2421 (2002)
7. M. Altun, M.D. Riedel, Robust computation through percolation: synthesizing logic with percolation in nanoscale lattices. Int. J. Nanotechnol. Mol. Comput. (IJNMC) **3**(2), 12–30 (2011)
8. A. Khitun, M. Bao, K.L. Wang, Spin wave magnetic nanofabric: a new approach to spin-based logic circuitry. IEEE Trans. Magn. **44**(9), 2141–2152 (2008)
9. Z. Abid, M. Liu, W. Wang, 3D integration of CMOL structures for FPGA applications. IEEE Trans. Comput. **60**(4), 463–471 (2011)
10. J. Von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components. Automata Stud. **34**, 43–98 (1956)
11. S. Gaba, P. Knag, Z. Zhang, W. Lu, Circuits and Systems (ISCAS), in *2014 IEEE International Symposium on Memristive devices for stochastic computing*. IEEE (2014, June), pp. 2592–2595
12. A.C. Arias, J.D. MacKenzie, I. McCulloch, J. Rivnay, A. Salleo, Materials and applications for large area electronics: solution-based ap-proaches. Chem. Rev. **110**(1), 3–24 (2010)
13. Y. Huang, X. Duan, Y. Cui, L.J. Lauhon, K.H. Kim, C.M. Lieber, Logic gates and computation from assembled nanowire building blocks. Science **294**(5545), 1313–1317 (2001)

14. W. Lu, C.M. Lieber, Nanoelectronics from the bottom up. Nat. Mater. **6**(11), 841–850 (2007)
15. P. Avouris, Molecular electronics with carbon nanotubes. Acc. Chem. Res. **35**(12), 1026–1034 (2002)
16. G. Snider, U.S. Patent No. 6,919,740. Washington, DC: U.S. Patent and Trademark Office (2005)
17. Z. Chen, et al., An integrated logic circuit assembled on a single carbon nanotube. Science 311.5768, 1735–1735 (2006)
18. H. Yan, H.S. Choe, S. Nam, Y. Hu, S. Das, J.F. Klemic, C.M. Lieber, Programmable nanowire circuits for nanoprocessors. Nature **470**(7333), 240–244 (2011)
19. C.P. Collier, E.W. Wong, M. Belohradský, F.M. Raymo, J.F. Stoddart, P.J. Kuekes, J.R. Heath, Electronically configurable molecular-based logic gates. Science **285**(5426), 391–394 (1999)
20. O. Tunali, M. Altun, Defect tolerance in diode, FET, and four-terminal switch based nano-crossbar arrays, in *IEEE International Symposium on Nanoscale Architectures, 2015. NANOARCH 2015*. pp. 82–87. IEEE (2015, July)
21. Y.C. Chen, S. Eachempati, C.Y. Wang, S. Datta, Y. Xie, V. Narayanan, Automated mapping for reconfigurable single-electron transistor arrays, in *Proceedings of the 48th Design Automation Conference,* pp. 878–883. ACM (2011, June)
22. Y. Levy, J. Bruck, Y. Cassuto, E.G. Friedman, A. Kolodny, E. Yaakobi, S. Kvatinsky, Logic operations in memory using a memristive Akers array. Microelectron. J. **45**(11), 1429–1437 (2014)
23. M. Altun, M.D. Riedel, Logic synthesis for switching lattices. IEEE Trans. Comput. **61**(11), 1588–1600 (2012)
24. A.M.S. Shrestha, S. Tayu, S. Ueno, Orthogonal ray graphs and nano-PLA design. In *ISCAS* (2009, May), pp. 2930–2933
25. W. Rao, A. Orailoglu, R. Karri, Topology aware mapping of logic functions onto nanowire-based crossbar architectures, in *Proceedings of the 43rd Annual Design Automation Conference*. ACM (2006, July), pp. 723–726
26. J. Huang, M.B. Tahoori, F. Lombardi, On the defect tolerance of nano-scale two-dimensional crossbars, in *Proceedings of 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2004. DFT 2004. IEEE (2004, October), pp. 96–104
27. M. Zamani, H. Mirzaei, M.B. Tahoori, ILP formulations for variation/defect-tolerant logic mapping on crossbar nano-architectures. ACM J. Emerg. Technol. Comput. Syst. (JETC) **9**(3), 21 (2013)
28. K. McElvain, in *Distributed as part of the MCNC International Workshop on Logic Synthesis IWLS'93 benchmark set: Version 4.0* (vol. 93) (1993, May)
29. B.R. Gaines (1967, April). Stochastic computing. In Proceedings of the April 18–20, 1967, Spring Joint Computer Conference. ACM, pp. 149–156
30. A. Alaghi, J.P. Hayes, Survey of stochastic computing. ACM Trans. Embed. Comput. Syst. (TECS) **12**(2s), 92 (2013)
31. W.S. Wong, A. Salleo (eds.), Flexible electronics: materials and applications (vol. 11). Springer Science & Business Media (2009)
32. S. Yavuz, M. Altun, Stokastik Hesaplamada Hata Oranlarını Azaltmak için Rastgele Bit Karıştırma Yöntemi—Random Bit Shuffling Method for Reducing Error Rates in Stochastic Computing, in *Elektrik—Elektronik—Bilgisayar ve Biyomedikal Mühendisliği Sempozyumu*, pp. 728–732. ELECO (2014)
33. R.A. Fisher, F. Yates, Statistical tables for biological, agricultural and medical research, 3rd edn. (1949)
34. R. Durstenfeld, Algorithm 235: random permutation. Commun. ACM **7**(7), 420 (1964)
35. H. Zhou, P.L. Loh, J. Bruck, The synthesis and analysis of stochastic switching circuits. (2012). arXiv preprint arXiv:1209.0715
36. C. Morgul, M. Altun, Synthesis and optimization of switching nanoarrays, in *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. pp. 161–164. IEEE (2015, April)