Fast Boolean Logic Mapped on Memristor Crossbar

Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Said Hamdioui, Koen Bertels
Laboratory of Computer Engineering,
Delft University of Technology, Delft, the Netherlands
Email: \{L.Xie,H.A.DuNguyen,M.Taouil,S.Hamdioui,K.L.M.Bertels\}@tudelft.nl

Abstract—As the CMOS technology is gradually scaling down to inherent physical device limits, significant challenges emerge related to scalability, leakage, reliability, etc. Alternative technologies are under research for next-generation VLSI circuits. Memristor is one of the promising candidates due to its scalability, practically zero leakage, non-volatility, etc. This paper proposes a novel design methodology for logic circuits targeting memristor crossbars. This methodology allows the optimization of the design of logic function, and their automatic mapping on the memristor crossbar. More important, this methodology supports the execution of Boolean logic functions within constant number of steps independent of its functionality. To illustrate the potential of the proposed methodology, multi-bit adders and multipliers are explored; their incurred delay, area and energy costs are analyzed. The comparison of our approach with state-of-the-art Boolean logic circuits for memristor crossbar architecture shows significant improvement in both delay (4 to 500×) and energy consumption (1.22 to 3.71×). The area overhead may decrease (down to 44%) or increase (up to 17%) depending on the circuit’s functionality and logic optimization level.

I. INTRODUCTION

As CMOS transistors gradually scale down to the inherent physical device limits, CMOS technology faces major challenges [1–4] such as increased leakage power consumption, saturated performance improvement, reduced reliability, and a more complicated fabrication process. To address these challenges, novel technologies (e.g., memristors [5,6], nanotube [7], graphene [8] and tunnel field-effect transistors [9], etc.) are proposed as the alternative for next-generation VLSI circuits. Among these technologies, memristor is a promising candidate [10,11]. Typically, numerous memristors can be mapped on crossbar architecture where memristors are located in the intersections of horizontal and vertical nanowires. Memristor crossbar is able to provide great scalability, higher integration density, zero leakage power consumption, CMOS fabrication compatibility [10,12–14]. Several potential applications have been proposed such as neuromorphic systems [15,16], non-volatile memories [10,17], novel computing paradigms for data-intensive applications [18], etc. To implement novel computing paradigms, it is crucial to design fundamental components such as Boolean logic functions [19].

Research on memristor-based logic circuits has attracted significant attention both in academia and industry since the first memristor device was fabricated by HP in 2008 [6]. Four types of logic circuits have been proposed: threshold [20,21], majority [21], material implication [22,23], and Boolean [24] logic. Threshold and majority logic circuits are based on threshold and majority logic gates, respectively; both of them are not suitable for crossbar arrays. However, both material implication and Boolean logic have been addressed for the memristor crossbar. In [23], the authors proposed a methodology to implement logic functions using a sequence of material implication operations. However, this methodology suffers from low speed and requires new algorithms to implement arithmetic operations such as addition and multiplication [23,25,26]. In [24], the authors proposed a systematic Boolean logic design methodology; its basic unit is a logic block which implements a Boolean function $f=M_1+\cdots+M_i+\cdots+M_n=M_1\cdots M_i\cdots M_n$, where $M_i$ denotes a minterm and $n$ is the total number of minterms. However, all the minterms are executed sequentially. As a consequence, building a complex logic circuit with numerous minterms will result in a very slow design. In addition, as each logic function requires different numbers of execution steps, the synchronization between different logic functions is a complicated task.

To solve aforementioned issues, this paper proposes a novel design methodology for crossbar-based Boolean logic circuits; it is based on parallel execution of all minterms and therefore significantly increases the overall performance. The contributions of this paper are:

- A methodology to design memristor logic circuits able to execute any Boolean function within constant time.
- Several novel primitive operations (implementing logic gates, copy operations, etc.) supporting the proposed design methodology.
- A novel memristor crossbar architecture supporting the proposed design methodology.
- A novel design technique to further optimize the proposed design methodology by sharing minterms.
- A model to evaluate the delay, area and energy.

The remainder of this paper is organized as follows. Section II briefly describes the background and related work. Section III proposes the design methodology for Boolean logic circuits. Section IV presents a one-bit full adder as a design case study. Section V verifies and evaluates the proposed design methodology. Finally, Section VI concludes the paper.
II. BACKGROUND AND RELATED WORK

This section describes the memristor model, data representation, control voltages, and the working principle of the previously proposed Boolean logic circuits [24].

A. Memristor Model

Although memristors have different physical mechanisms [12,13,27], their circuit-level behaviour can be abstracted by two parameters: switching threshold voltage $V_{th}$ (with or without a minimum voltage requirement for resistive switching) [28] and resistive switching behaviour (abrupt or smooth) [29]. Note that these two parameters are independent of each other.

In this paper, we use a simplified memristor model based on a switching threshold voltage and abrupt switching [23,24]. For simplicity, this type of model is referred to as THAB (THreshold-ABrupt) memristor model. The top part of Fig. 1(a) illustrates the current-voltage relation of THAB model. The memristor switches from one resistive state to another when the absolute value of the voltage (either positive or negative) across the device is greater than its threshold voltage $V_{th}$; see I-V curve of Fig. 1(a). Otherwise, it stays in its current resistive state. Normally, a memristor requires two different switching threshold voltages to switch from low to high resistance (RESET) and from high to low resistance (SET) [28,30]; see the bottom part of Fig. 1(a). The black squares represent the positive terminal of the memristor. For simplicity, we assume that THAB model has the same threshold voltage $V_{th}$ (in absolute value) for both of them.

B. Data Representation and Control Voltages

A memristor has two resistive states: a high ($R_{on}$) and low ($R_{off}$) resistance. In this paper, all logic circuits use $R_{off}$ and $R_{on}$ for logic 1 and 0, respectively. It is worth to note that this is different from CMOS logic circuits which use high and low voltages to represent logic 1 and 0, respectively.

To control memristor-based digital circuits, three different voltages are required: $V_w$, $V_{wh}$, and GND; see Fig. 1(a). $V_w$ is used to program the resistance of a memristor; $V_{wh}$ is used to minimize the impact of sneak path currents by half-select voltage strategy [17]; $V_{wh}$ is then applied to memristors which are not involved in particular operations. $V_{wh}$ is also used to support the implementation of basic logic gate as in the case of NAND [24]. The relationship between $V_w$, $V_{wh}$, GND and $V_{th}$ is $0 < V_{wh} < V_{th} < V_w$. This relationship guarantees $V_w - V_{wh} = 2V_{wh} = V_{wh} < V_{th}$ which is able to prevent undesired switching of non-accessed memristors [17].

C. Snider’s Boolean Logic Circuits

G. Snider proposed in [24] a design methodology for memristor crossbar to implement Boolean logic. In this paper, we denote this design methodology by “SBLC” (Snider’s Boolean Logic Circuits). As Fig. 1(b) shows, SBLC is formed by an alternating cascade of latches and logic blocks; synchronized by a CMOS control logic (realized by a FSM). Each Boolean function is described as the format of $f = M_1 + \cdots + M_i + \cdots + M_n = M_1 \cdot \cdots \cdot M_i \cdots \cdot M_n$, and is implemented by a basic computing element (CE) as shown in the bottom of Fig. 1(b). A CE consists of an input latch (IL), an output latch (OL) and a logic block (LB). The LB consists of all the minterms of the Boolean function; each $M_i$ is realized using a NAND gate consisting of several memristors based on its number of inputs. The OL of a particular CE is connected to the IL of the next CE from its OL. The IL of the CE in the case of NAND [24]. The relationship between $V_w$, $V_{wh}$, GND and $V_{th}$ is $0 < V_{wh} < V_{th} < V_w$. This relationship guarantees $V_w - V_{wh} = 2V_{wh} = V_{wh} < V_{th}$ which is able to prevent undesired switching of non-accessed memristors [17].

SBLIC requires an appropriate control to execute the steps needed to compute a Boolean function. This is captured by the state machine in Fig. 1(c); $n_p$ represents the number of minterms processed at each step, $n$ the total number of minterms. The state machine consists of six states:

1) INA: Iniitalize All the memristors of a CE to $R_{off}$.
2) RI: Receive Inputs. The IL of the CE receives the outputs of the previous CE from its OL. The IL of the CE in the first stage receives the inputs from primary inputs; e.g., A.
3) CFM: ConFigure a Minterm. A particular minterm $M_i$ is configured by copying data from IL. Simultaneously,
the memristor located at the intersection of the column implementing the minterm \( M_i \) and row of the output is SET. This forms a low-resistance path between IL, \( M_i \) and OL.

4) EVM: Evaluate a Minterm. The \( \overline{M_i} \) is implemented by a NAND gate and its result is accumulated at OL.

5) RSM: Reset a Minterm. All the memristors of \( M_i \) are RESET.

6) SO: Send Outputs. The results of all the outputs stored in OL are sent to IL of the next CE.

Each minterm requires three steps: CFM, EVM and RSM. The CE repeats these three steps until the minterms are evaluated. In conclusion, a CE with \( n \) minterms requires \( 3n+3 \) steps to compute the Boolean function. This sequential execution induces slow speed, and requires complex synchronization between multiple LBS.

### III. Logic Circuits Design Methodology

This section describes the proposed design methodology, its primitive operations, and how the crossbar architecture should be tuned to make it suitable for the proposed method.

#### A. Working Procedure and Primitive Operations

The major drawback of SBLC is the sequential execution of minterms; this is due to two reasons. First, the minterms cannot copy data from the input latches simultaneously. Second, all the minterms accumulate their results in the same output latch to produce the final result.

Motivated by these two reasons, we propose fast Boolean logic circuit (FBLC) which requires the Boolean function to be expressed in the sum-of-product format; i.e., \( f=M_1+\cdots+M_i+\cdots+M_n=\overline{M_1}\cdot\cdots\cdot\overline{M_i}\cdot\cdots\cdot\overline{M_n} \). The concept of the proposed method is shown in Fig. 2(a). Different from SBLC, FBLC employs a novel primitive operation that allows all the minterms to copy their data simultaneously from the input latch. In addition, all the \( \overline{M_i} \) are evaluated (based on NAND operations), and subsequently the results of \( \overline{M_i} \) are processed by an AND operation to obtain the final result; this result is stored in OL. As a result, FBLC can process all the minterms in parallel, and therefore a Boolean function can be calculated in a constant number of steps independent of the circuit’s functionality as will be explained later in this section. It is worth to note that FBLC uses both primary and complementary inputs as CEs in the next stage typically require both.

FBLC requires 7 steps to compute a Boolean function independent of its functionality. Fig. 2(b) shows the state machine that generates the control signals for FBLC. The required primitive operations of each state are:

- **INA**: Initialize all the memristors of a CE to \( R_{off} \). This state requires RESET operation.
- **RI**: Receive Inputs. The IL of the CE receives the OL of the previous CE. The IL of the first stage CE receives the inputs from primary inputs. Therefore, this state requires single-fan-out copy (SFC) operation.
- **CFM**: Configure all Minterms. All the minterms are configured simultaneously by copying inputs stored in IL to each minterm in parallel. Hence, this state requires multi-fan-out copy (MFC) operation.
- **EVM**: Evaluate all Minterms. All the \( \overline{M_i} \) are evaluated simultaneously; again, the implementation is based on an NAND operation. Hence, this state requires NAND operations: a basic single-fan-out NAND (SFNAND) used to evaluate the \( \overline{M_i} \), and an advanced multi-fan-out NAND (MFNAND) used to further optimize FBLC; the latter will be discussed in section IV.
- **EVR**: Evaluate Results. The results of EVM are fed to an AND gate to determine \( f \) of the Boolean function; see Fig. 2(a). Therefore, this state needs an AND operation.
- **INR**: Invert Results. The result of EVR need to be inverted to achieve the final result \( f \) of the Boolean function. Hence, an inversion (INV) operation is required.
- **SO**: Send Outputs. Finally the result captured in OL is sent to IL of the next CE. Hence, a SFC is needed.

The left part of Table I summarizes the primitive operations required at each state; the right part of the table presents the implementation of primitive operations which will be discussed in the next subsection. The first column shows the states; the second column the required operations. Column 3 and 4 present the logic values of the inputs and outputs, respectively; the logic value of a single-input operation is either 1 or 0. For multi-input operations, we distinguish between A1 (all inputs are 1) and AL0 (at least one input is 0).

---

**TABLE I: Summary of Primitive Operations**

<table>
<thead>
<tr>
<th>State</th>
<th>Operation</th>
<th>Logic Value</th>
<th>Control &amp; Intermediate Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>INA</td>
<td>RESET</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RI</td>
<td>SFC</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CFM</td>
<td>MFC</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EVM</td>
<td>SFNAND</td>
<td>A1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MFNAND</td>
<td>AL1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>EVR</td>
<td>AND</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AL0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>INR</td>
<td>INV</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SO</td>
<td>SFC</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 2: Fast Boolean Logic Circuits.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>( f )</th>
<th>( \overline{f} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_w )</td>
<td>( GND )</td>
<td>0 &gt; ( V_h )</td>
<td>0 &lt; ( V_h )</td>
</tr>
<tr>
<td>( V_w )</td>
<td>( V_w )</td>
<td>0 &gt; ( V_h )</td>
<td>0 &lt; ( V_h )</td>
</tr>
<tr>
<td>( GND )</td>
<td>( V_w )</td>
<td>0 &gt; ( V_h )</td>
<td>0 &lt; ( V_h )</td>
</tr>
<tr>
<td>( V_w )</td>
<td>( V_w )</td>
<td>0 &gt; ( V_h )</td>
<td>0 &lt; ( V_h )</td>
</tr>
<tr>
<td>( GND )</td>
<td>( V_w )</td>
<td>0 &gt; ( V_h )</td>
<td>0 &lt; ( V_h )</td>
</tr>
<tr>
<td>( V_w )</td>
<td>( V_w )</td>
<td>0 &gt; ( V_h )</td>
<td>0 &lt; ( V_h )</td>
</tr>
</tbody>
</table>

366 2015 33rd IEEE International Conference on Computer Design (ICCD)
B. Implementation of Primitive Operations

Fig. 3 shows the implementation of each primitive operation consisting of one or multiple input and output memristors. The output memristors are surrounded by a box with a dashed-line and are all initialized to \( R_{off} \) prior to any operation (i.e., RESET operation of state INA). In addition, the implementation of each operation consists of a resistor \( R_s \), which satisfies the condition \( R_{on} \ll R_s \ll R_{off} \); this is required to guarantee that the voltage across the output memristors are close to the desired voltage (e.g., \( V_w \approx V_{wh} \)) for appropriate operations [23].

The right part of Table I shows the control and intermediate signals (for each primitive operation) related to the implementation. Column ‘\( V_{ci} \)’ and ‘\( V_{co} \)’ present the control voltages applied to the input and output memristors, respectively. Column ‘\( V_x \)’ presents the voltage of the floating nanowires for given inputs of a particular operations; it is the voltage across \( R_s \) and is used to explain the working principle; see Fig. 3. Column ‘\( V_{om} \)’ shows the voltage across the output memristor after all the control voltages are applied. It indicates whether the output memristor remains \( R_{off} \) (\( V_{om} < V_{th} \)) or switch to \( R_{on} \) (\( V_{om} > V_{th} \)).

The primitive operations are implemented as follows:

- **State INA**: State INA requires RESET operation; its working principle has been explained in Fig. 1(a).
- **State RI**: State RI requires SFC operation; its working principle is shown in Fig. 3(a). A voltage \( V_w > V_{th} \) (see Fig. 1(a)) is applied to the negative terminal of the input memristor while \( GND \) is applied to the output memristor. In case the input is 1 (\( R_{off} \)), \( V_z \approx 0 \) as \( R_s \ll R_{off} \) (see Fig. 3(a)-Copy 1). Therefore, \( V_{om} = V_z - 0 \approx 0 < V_{th} \). As a result, the output memristors stay at \( R_{off} \). In case the input is 0, \( V_z \approx V_w \) as \( R_{on} \ll R_s \) (see the left part of Fig. 3(b)-Copy 0). Therefore, \( V_{om} = V_z - 0 \approx V_w > V_{th} \). As a result, the output memristors switch to \( R_{on} \). However, this operation is destructive as shown in the right part of Fig. 3(b)-Copy 0. After output memristors switch to \( R_{on} \), \( V_z \) changes from \( V_w \) to 0. Therefore, the voltage across the input memristor becomes \( V_z - V_{wh} \approx V_z - V_{th} \). As a result, the input memristor switches to \( R_{off} \). Nevertheless, this destructive operation has no impact on the correctness of the CE as the data stored in IL is correctly copied to all minterms.
- **State EVM**: State EVM requires SFNAND operation; its working principle is shown in Fig. 3(c). A voltage \( V_{wh} < V_{th} \) (see Fig. 1(a)) is applied to the positive terminal of the input memristor while \( V_w \) is applied to the output memristor. In case all the input are 1, \( V_z \approx 0 \) (see Fig. 3(c)) and \( V_{om} = V_w - V_z \approx V_w > V_{th} \). As a result, the output memristor switches to \( R_{on} \). In case at least one input is 0, \( V_z \approx V_{wh} \) and \( V_{om} = V_w - V_z \approx V_w - V_{wh} < V_{th} \). As a result, the output memristor stays at \( R_{off} \).
- **State EVR**: State EVR requires AND operation; its working principle is shown in Fig. 3(e). \( V_w \) is applied to the negative terminal of the input memristors while \( GND \) is applied to the output memristor. In case all the inputs are 1, \( V_x \approx 0 \) and \( V_{om} = V_z - 0 \approx 0 < V_{th} \). As a result, the output memristor stays \( R_{off} \). In case at least one input is 0, \( V_x \) is around \( V_w \) and \( V_{om} = V_z - 0 \approx V_w - V_{wh} > V_{th} \). As a result, the output memristor switches to \( R_{on} \).
- **State INR**: State INR requires INV operation; its working principle is shown in Fig. 3(f). \( V_{wh} \) is applied to the positive terminal of the input memristor while \( V_w \) is applied to the output memristor. In case the input is 1, \( V_z \approx 0 \) as \( R_{on} \ll R_{off} \) (see Fig. 3(f)-Invert 1). Therefore, \( V_{om} = V_w - V_z \approx V_w > V_{th} \). As a result, the output memristor switches to \( R_{on} \). In case the input is 0, \( V_z \approx V_{wh} \) as \( R_{on} \ll R_s \) (see Fig. 3(f)-Invert 0). Therefore, \( V_{om} = V_w - V_z \approx V_w - V_{wh} < V_{th} \). As a result, the output memristor stays at \( R_{off} \).
- **State SO**: State SO requires the same SFC operation the one used in state RI.
A. Case Study: One-Bit Full Adder

The sum and carry of a one-bit full adder (FA) can be expressed by Eq. 1. Each expression consists of four minterms. The FBLC implementation of this 1-bit FA can be mapped to CMCA as shown in Fig. 4 (a). For convenience, H# and V# are used to denote a horizontal and vertical nanowire, respectively. We present the position of a memristor by P(H#,V#). This 1-bit FA can be implemented using a CE consisting of an IL, LB and OL. The IL is mapped on the memristors located at P(H1,V1-V6) as IL consists of primary inputs and their complements. The remaining memristors on H1 are disabled. The LB consisting of eight minterms is mapped on H2-H9. Each minterm is implemented by placing active memristors at junctions/positions formed by the horizontal nanowire (representing the minterm) and (a) the vertical nanowires associated with the minterm’s inputs, or (b) an output for which the minterm is part of. For example, ABC on H2 is a minterm of Sum. Therefore, the junctions on H2 at P(H2,V2=\overline{A}), P(H2,V4=\overline{B}), P(H2,V5=C) as well as at P(H2,V7=\overline{Sum}) consist of active memristors; while the remaining junctions on H2 contain disabled memristors (non-active). The four minterms of \textit{Sum} and those of \textit{Carry} (see Eq. 1) are then ANDed in parallel by column V7=\overline{Sum} and V8=\overline{Carry}, respectively. The OL is realized by H10 and H11. The results provided by the two ANDs are then stored at P(H10=\overline{Sum}, V7) and P(H11=\overline{Carry}, V8), which are thereafter inverted and stored at P(H10, V9=\overline{Sum}) and P(H11, V10=\overline{Carry}), respectively.

\[
\textit{Sum} = ABC \cdot \overline{ABC} \cdot \overline{ABC} \cdot ABC
\]

\[
\textit{Carry} = ABC \cdot \overline{ABC} \cdot \overline{ABC} \cdot ABC
\]

IV. DESIGN CASE STUDY AND OPTIMIZED FBLC

This section presents a one-bit full adder based on FBLC as a case study. Subsequently, it presents how to further optimize FBLC and a generic mapping process that maps arbitrary Boolean functions to the crossbar.
TABLE II: Control Voltages for One-Bit Full Adder

<table>
<thead>
<tr>
<th>State</th>
<th>InA</th>
<th>InB</th>
<th>InC</th>
<th>R lun</th>
<th>EVM</th>
<th>EVR</th>
<th>INR</th>
<th>state</th>
<th>InA</th>
<th>InB</th>
<th>InC</th>
<th>R lun</th>
<th>EVM</th>
<th>EVR</th>
<th>INR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Sum</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Carry</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Carry</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Sum</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Carry</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Carry</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 5: Mapping Process for an Arbitrary Boolean Function

nanowires, the related memristors perform the specific primitive operations which together execute a Boolean function. The applied voltages to each horizontal and vertical nanowire of the 1-bit FA during each state are summarized in Table II; they are derived from Table I. PSOL and NSIL represent the OL of the previous stage and the IL of the next stage, respectively. For instance, to perform MFC operation (see Fig. 3(b)) required by CFM state in order to configure all minterms in parallel, $V_{wh}$ ($V_{wh}$ in the row CFM of Table I) is applied to horizontal nanowire H1 while GND ($V_{wh}$ in the row CFM of Table I) is applied to horizontal nanowires H2-H9; vertical nanowires V1-V6 are floating (F) ($V_{wh}$ in the row CFM of Table I). $V_{wh}$ is applied to all the other horizontal and vertical nanowires in order to minimize the impact of sneak path currents [31].

B. Optimized Fast Boolean Logic Circuits

We observed that the nanowires H8 and H9 in Fig. 4 (a) compute the same minterm $ABC$ but for the different outputs $Sum$ and $Carry$, respectively. Sharing this minterm would reduce both area and energy consumption. Based on this, we introduce a minterm-sharing strategy. Outputs with a common minterm will be connected to a shared nanowire and employ a MFNAND instead of SFNAND. The MFNAND operation works similarly as SFNAND except that the results are obtained in multiple output memristors as shown in Fig. 3(d). Fig. 4(b) shows the result after applying the minterm-sharing strategy to the 1-bit FA of Fig. 4(a); $ABC$ is now presented only by H8. FBLC with the minterm-sharing is referred to as optimized FBLC (OFBLC).

C. Generic Mapping Process

The implementation of the 1-bit FA can be generalized and any Boolean function can be mapped on CMCA by using its truth table as shown in Fig. 5. First, an initial truth table is created based on the Boolean function. This truth table can be further optimized using existing logic optimization tools (e.g., ESPRESSO [32]). Next, minterms are extracted from the (optimized) truth table. Finally, the minterms are mapped on the crossbar, similarly as shown for the 1-bit FA.

V. RESULTS AND EVALUATION

This section first verifies the 1-bit FA using SPICE simulation; thereafter, it describes the evaluation metrics and comparison between SBLC, FBLC and OFBLC.

TABLE III: Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{on}$</td>
<td>100Ω</td>
<td>$V_{wh}$</td>
<td>1.4V</td>
</tr>
<tr>
<td>$R_{off}$</td>
<td>200kΩ</td>
<td>$V_{th}$</td>
<td>0.7V</td>
</tr>
<tr>
<td>$R_a$</td>
<td>1kΩ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. One-Bit Full Adder Verification

The one-bit full adder based on OFBLC shown in Fig. 4(b) is verified by SPICE simulations. The memristor model, control logic and voltage drivers are described by Verilog-A modules, while the memristor crossbar circuit by a SPICE netlist. The technology parameters and control voltages are listed in Table III. The logic values 1 and 0 are represented by $INR$ and $INV$, respectively, to perform $MFC$ operation (see Fig. 3(b)) required by $CFM$ state in order to configure all minterms in parallel, $V_{wh}$ ($V_{wh}$ in the row CFM of Table I). $R_{on}$ and $R_{off}$, respectively. As only a single CE is simulated, the inputs are directly programmed in state RI (instead of copying data from a previous CE) and state SO is ignored.

All the 8 possible input combinations of the 1-bit FA have been
verified as shown in Fig. 6. The figure shows for each state the resistance values of the inputs (i.e., \(A, \overline{A}, B, \overline{B}, C\) and \(\overline{C}\); see Fig. 4(b) at \(P(H1,V1-V6)\)) and outputs (i.e., \(Sum, Carry, Sum\) and \(Carry\); see Fig. 4(b) at \(P(H9,V7), P(H10,V8), P(H9,V9)\) and \(P(H10,V10)\), respectively). The inputs of the 1-bit FA are programmed during state RI while the outputs are valid at state INR. For instance, Fig. 6(a) shows the case the inputs of the adder are all zero \((ABC=000)\). Its corresponding outputs \((Sum=1,Carry=1,Sum=0, Carry=0)\) are visible at state INR.

### B. Evaluation Metrics

The evaluation metrics delay, area and energy consumption are used to compare the performance between SBLC and (O)FBLC.

The delay is defined as the number of steps required to execute a CE; each step, executed by a state, is assumed to complete in one cycle. For a fair comparison, we add an extra INR state to SBLC to compensate for the complementary outputs. The delay of a CE that consists of \(n_m\) minterms requires \(3n_m+4\) cycles for SBLC while 7 cycles for (O)FBLC as summarized in Table IV.

Fig. 7 shows the layout dimensions of a generic CE for SBLC, FBLC and OFBLC. In the figure, \(n_i\) presents the number of inputs, \(n_o\) the number of outputs, \(n_m\) the total number of minterms without considering minterm-sharing, and \(n_{ms} (n_{ms}\leq n_m)\) the total number of minterms with minterm-sharing. The calculated area is shown in Table IV and includes both active and disabled memristors.

The energy consumption of a CE for a specific input combination is dominated by the resistive switching activity (from high-to-low and low-to-high), as memristors have near-zero leakage power consumption [10]. We assume that both high-to-low and low-to-high resistive switching consume the same amount of energy \(E_{sw}\). In addition, we assume that each input combination occurs with the same probability. The average energy consumption of all the input combinations is used as energy metric.

### C. Comparison

To compare (O)FBLC with SBLC, we designed CEs that implement 2- and 4-bit full adders and multipliers. Based on the generic mapping process of Fig. 5, we developed automated scripts that map the adders and multipliers on the crossbar and subsequently evaluate their delay, area, and energy consumption. Both initial and optimized (using ESPRESSO [32]) truth tables are used for comparison. The results are shown in Table V. SBLC with the initial truth table as input is utilized as baseline for comparison. The results of (O)FBLC are normalized to this baseline which is denoted by the improvement factor (IF).

From the table we conclude the following:

- For SBLC designs, the delay increases significantly when the number of minterms increases. In contrast, the delay of (O)FBLC designs is constant (7 cycles) for all the four cases and up to 500 times faster than those of SBLC designs.
- The area of OFBLC designs is typically smaller than FBLC designs (up to 66% area improvement), while FBLC designs perform area-wise always worse than SBLC (up to 31% area increase). However, the area of OFBLC designs can be larger or smaller than those of SBLC designs depending on whether ESPRESSO is used or not. In case ESPRESSO is not used, we observe that the area of OFBLC designs is smaller than those of SBLC designs (up to 56% area decrease) due to the minterm sharing. In case ESPRESSO is used, we observe that the area of OFBLC designs is typically larger than those of SBLC designs (up to 17% area increase).
- OFBLC designs consume in all cases the lowest energy as they have the least number of minterms, and therefore the least number of switching memristors. In contrast, SBLC designs perform worst; their energy consumption is 1.36 to 3.71 times higher than those of OFBLC designs.
- In case ESPRESSO is not used, OFBLC designs are able to reduce the number of minterms significantly with respect to SBLC and FBLC designs due to minterm sharing. Therefore, OFBLC designs perform best in all these three aspects. However, the benefit of minterm sharing reduces when the logic optimization (i.e., ESPRESSO) is used.

The results of the experiments clearly show that although (O)FBLC designs have typically a slight penalty in terms of area with respect to SBLC designs, but it has a significant improvement both in delay and energy consumption.

### VI. Conclusion

Boolean logic circuits based on memristor crossbar are a promising candidate for the next-generation VLSI circuits. This paper proposed fast Boolean logic circuits (FBLC) for memristor crossbars able to execute any Boolean function within constant time. Comparing with state-of-the-art memristor-based Boolean logic circuits, FBLC has significant improvement in terms of delay and energy consumption. Moreover, FBLC provides a systematic design methodology starting from Boolean functions all the way down to memristor crossbar implementations. This process is compatible with existing logic optimization tools (e.g., ESPRESSO), and...
therefore, feasible to be integrated into existing design flows. Based on the above features, FBLC can be used as one of the candidates to design basic building blocks in memristor-based VLSI circuits.

References


