Factored Forms for Memristive Material Implication Stateful Logic

Felipe S. Marranghello, *Student Member, IEEE*, Vinicius Callegaro, *Student Member, IEEE*, Mayler G. A. Martins, *Student Member, IEEE*, André I. Reis, *Senior Member, IEEE*, and Renato P. Ribas, *Member, IEEE*

***Abstract*—This paper proposes the utilization of factored forms in logic synthesis for memristive material implication stateful logic. Factored forms have not been explored by previous works due to expected increasing on device count. We present an algorithm to obtain factored forms computable with minimum number of memristors. Comparison to previous works shows an average reduction of 12% in the number of operations to compute 4-input Boolean functions.**

***Index Terms*—Digital circuit, implication logic, logic synthesis, memristor, stateful logic.**

I. Introduction

MEMRISTORS are two-terminal passive device firstly theorized in 1971 as a basic circuit element [1], [2]. Even though memristive behavior has been observed for a long time [3], the relationship between theory and experimental results was only demonstrated in 2008, with the fabrication of the first nanoscale memristor device [4]. Moreover, memristors had already been implemented using transistors and capacitors, but such implementations were not compact enough [5]. The nanoscale memristor renewed the interest in using such a device on VLSI design. Recently, several works have studied the dynamic behavior of memristors [6]–[9] as well as the utilization of this device to design memories [10]–[13], neuromorphic systems [14]–[18], analog [19]–[21] and digital circuits [22]–[34].

There are different manners to exploit memristors in digital circuit design. They can be used together with MOS transistors to implement threshold logic gates [22]–[25] and as reconfigurable switches [26]–[28]. In both cases, Boolean values are represented by voltage levels. Alternatively, the logic information can also be represented through the memristor resistance value. Such a strategy allows the implementation of stateful logic, i.e., logic elements are also used for storaging [15], [29]–[34].

The material implication (IMPLY) based architecture, described in [30], is the most adopted approach to perform stateful logic. Using this architecture, any n-input Boolean function, expressed in a recursive form, can be computed with $n + 2$ memristors [35]. On the other hand, computation based on this architecture is typically sequential. Consequently, the computation time is directly related to the number of operations performed. In this sense, methods to improve performance of memristive IMPLY stateful logic are of great interest. However, these methods may increase the number of devices required to perform logic computation. For instance, $n + 2$ memristors blocks can be used in parallel, where each block computes a subfunction of the target function [36].

Performance improvement can also be obtained by reducing the number of operators in IMPLY expressions. In this sense, a *multi-input implication* operation performs a logical disjunction among several inputs in a single step [37], [38]. Logic synthesis methods have exploited such an operation to improve quality of IMPLY expressions [39], [40]. These methods only synthesize recursive forms to ensure that the resulting expressions are computable with $n + 2$ memristors.

This paper proposes the utilization of factored forms for memristive IMPLY stateful logic. These forms are more general than the recursive forms presented in [35], leading to shorter IMPLY expressions. However, most factored forms are not computable with $n + 2$ memristors. In this work, we present an algorithm to obtain factored forms computable with $n + 2$ memristors. Experimental results considering all 4-input Boolean functions show significant reduction on the average number of IMPLY operations. To achieve such result, two other contributions are also presented: 1) we demonstrate that a simple logic simplification applied in previous works should be avoided, and 2) the concept of multi-input implication is generalized to *multi-memristor implication*.

The rest of the paper is organized as follows. Section II briefly discusses the electrical behavior of memristors. Section III reviews the memristive IMPLY stateful logic. Section IV shows that a simple logic simplification adopted by previous algorithms is not effective. Section V presents the multi-memristor implication concept. Section VI evaluates the computation of factored forms. Section VII describes a new algorithm to synthesize Boolean functions, considering different types of expressions. Section VIII presents experimental results and Section IX outlines the conclusions.

II. Memristor

The memristor is a two terminal passive device that behaves as programmable resistor (memristor is short for memory resistor) [1]. The device resistance can be modified by applying a

Manuscript received November 07, 2014; revised February 02, 2015; accepted March 23, 2015. Date of publication May 15, 2015; date of current version June 09, 2015. Research funded by the Brazilian funding agencies CAPES, CNPq and FAPERGS, under Grant 11/2053-9 (Pronem). This work was supported T. Prodromakis.

The authors are with the PGMicro/PPGC, Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), 9500 Porto Alegre, Brazil (e-mail: fsmarranghello@inf.ufrgs.br; vcallegaro@inf.ufrgs.br; mga-martins@inf.ufrgs.br; andreis@inf.ufrgs.br; rpribas@inf.ufrgs.br).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JETCAS.2015.2426511

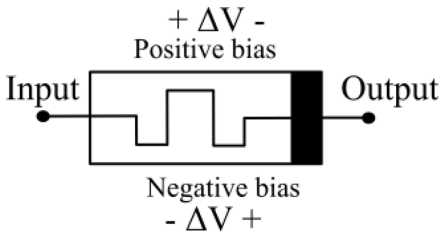

Fig. 1. Memristor electrical symbol. Output terminal is marked by a thick black line. A positive (negative) bias reduces (increases) the resistance value.

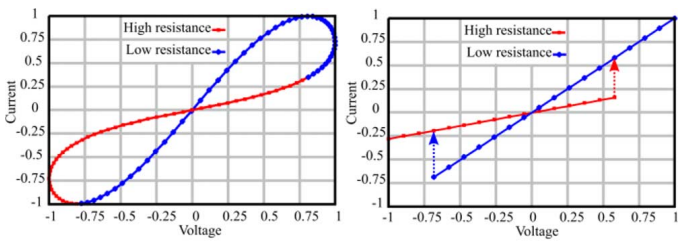

Fig. 2. Current-voltage memristor behavior: (a) linear and (b) nonlinear.

voltage difference between its terminals. If a positive voltage bias is applied, then the device resistance value tends to decrease. In opposite, when a negative voltage bias is applied, the resistance value increases. The resistance remains unaltered without a voltage biasing. Fig. 1 illustrates the memristor electrical symbol.

The memristor electrical behavior is highly dependent on the device material and fabrication process. [41]. Therefore, different types of memristor devices have been proposed. One possible classification regards the dependence of the dopant drift velocity with respect to the applied electrical field, which can be either linear or nonlinear [42]. Linear drift devices present a smooth transition between high and low resistance states, whereas nonlinear drift devices present abrupt transitions between the states, being possible the identification of threshold voltages (one positive and one negative). Fig. 2 illustrates the current-voltage behavior for linear and nonlinear memristors. Nonlinear devices are preferred to implement stateful logic because of nondestructive reads [42]. Hereafter, only nonlinear devices are taken into account.

III. Memristive Imply Stateful Logic

A. Notation

An n-input Boolean function $F(X)$ defined over the variable set $X = \{x_1, \ldots, x_n\}$ is a function $F(X) = B^n \mapsto B$, where $B = \{0, 1\}$. The constant functions zero and one are denoted by 0 and 1, respectively. In this work, we use f^* to represent a Boolean expression for F over the basis $\{\wedge, \vee, \neg\}$, where $\{\wedge, \vee, \neg\}$ denote logical conjunction, disjunction and complementation, respectively. Alternatively, the notations $x^0 = x \rightarrow 0 = \neg x$ and $x^1 = x$ are also adopted. Finally, f denotes an IMPLY based expression for F.

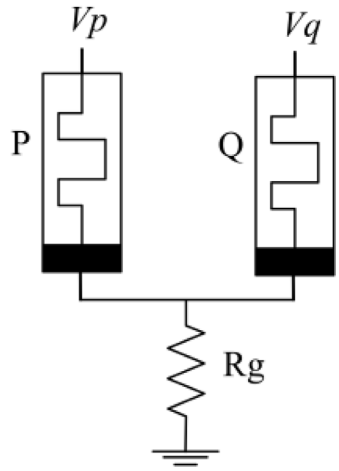

Fig. 3. Basic memristive IMPLY stateful logic structure [30].

TABLE I
Material Implication Truth ($p \rightarrow q$)

p	q	*next q*
0	0	1
0	1	1
1	0	0
1	1	1

B. Single Memristor

Consider a single memristor, as depicted in Fig. 1, where the output terminal is connected to ground reference and the input terminal voltage can assume three different values: *Vset, Vclear* or *Vcond*. *Vset* (*Vclear*) is a voltage greater (smaller) than the memristor positive (negative) threshold, forcing the device into the low (high) resistance state. *Vcond* is a positive voltage smaller than the positive threshold, having negligible influence on the device state.

C. Basic Structure

The basic structure for memristive IMPLY stateful logic is depicted in Fig. 3 [30]. Two memristors P and Q, controlled by voltages *Vp* and *Vq*, interact through a common node connected to a load resistor Rg. Each memristor is a bipolar device with memristance M. When $M = M_{\mathrm{ON}} (M = M_{\mathrm{OFF}})$, the memristor is in the low (high) resistance state and stores the logic one (zero). The binary states of P and Q are given by p and q, respectively.

The common node makes the voltage difference between the terminals of memristor P (Q) dependent on *Vq (Vp)*. Consider that *Vcond* is applied to *Vp* while *Vset* is applied to *Vq*. If P is in high resistance state ($p = 0$), then *Vp* has negligible influence on the circuit, and *Vset* is able to set Q to the low resistance state ($q = 1$). On the other hand, if P is in the low resistance state ($p = 1$), then *Vp* increases the voltage across Rg, resulting in a voltage drop in Q insufficient to modify the value of q. Hence, q remains with the previous value. Table I shows the final values of q (*next q* column) as function of the initial values p and q. The logic behavior shown in Table I corresponds to the material

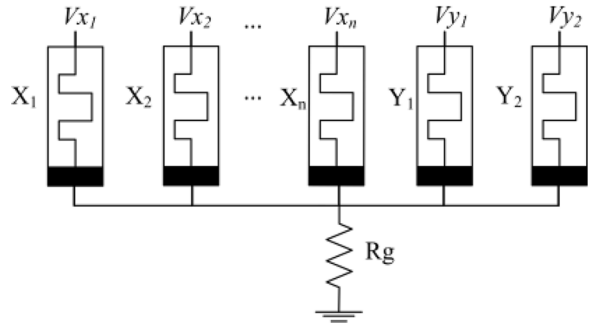

Fig. 4. Memristor based stateful logic gate.

implication function ($\rightarrow$). Every time an operation $p \rightarrow q$ is performed, the resulting value is stored in Q, overwriting the initial value q.

The correct functioning of this circuit depends on the values chosen for control voltages (*Vp* and *Vq*) and the load resistance (Rg). Such electrical challenges are extensively discussed in the literature [43], [44]. In this work, we focus on the logic behavior of the circuit.

D. General Structure

The general circuit to compute an n-input Boolean function is shown in Fig. 4, where devices X_1 to X_n are input memristors and devices Y_1 and Y_2 are work memristors. Each input memristor stores the value of an input variable, and the work memristors are used to keep intermediate computation values. For each memristor $\mathrm{X}_\mathrm{k}(Y_\mathrm{k})$, its control voltage is denoted by $Vx_k(Vy_k)$ and its state is denoted by $x_k(y_k)$, representing a Boolean variable. Logic computation is performed in the same way as in the circuit shown in Fig. 3. To perform a IMPLY operation $x_k \rightarrow y_j$, *Vcond* is applied to X_k while *Vset* is applied to Y_j. Input nodes of remaining memristors are set to high-impedance, preventing undesirable changes of memristors states.

A reset operation programs a work memristor Y_k to the high resistance state by applying *Vclear* to the target device. This operation is denoted by $y_k = 0$. Usually, a reset operation is performed before a complementation operation on a variable $x_k(x_k \rightarrow 0)$ to ensure that the target memristor stores the logic zero. Notice that reset operations do not explicitly appear on IMPLY expressions.

1) Example 1: Consider a Boolean function F with input variables x_1 and x_2, which are stored in memristors X_1 and X_2, respectively. Y_1 and Y_2 are the work memristors. Let F be represented by

$$f^* = x_1 \wedge \neg x_2. \tag{1}$$

An IMPLY expression for F is the following:

$$f = (x_1 \rightarrow ((x_2 \rightarrow 0) \rightarrow 0)) \rightarrow 0. \tag{2}$$

Equation (2) can be compute in the four following steps.

Step 1: This step performs $x_2 \rightarrow 0$. To ensure that the state of Y_1 is zero, a reset operation is performed on Y_1. Then, *Vcond* is applied to memristor X_2 and *Vset* to Y_1. Y_1 stores the complementary value of $x_2(\neg x_2)$. The operations performed are: $y_1 = 0$ and $x_2 \rightarrow y_1$.

Step 2: This step writes x_2 into work memristor Y_2. Reset Y_2. Apply *Vcond* to Y_1, and *Vset* to Y_2. The operations performed are: $y_2 = 0$, $y_1 \rightarrow y_2$.

Step 3: This step performs a material implication from X_1 to Y_2. Apply *Vcond* to X_1 and *Vset* to Y_2. The operation performed is $x_1 \rightarrow y_2$.

Step 4: This step performs the complementation operation on Y_2. This result must be kept on Y_1, which keeps the value $\neg x_2$ from Step 1. Hence, the state y_1 must be reset. After the reset operation, apply *Vcond* to Y_2 and *Vset* to Y_1. Y_1 stores the resulting value of (2). The performed operations are: $y_1 = 0$ and $y_2 \rightarrow y_1$. Table II presents the states of work memristors at the end of each step when computing (2).

TABLE II
WORK MEMRISTORS STATES AT EACH STEP WHEN COMPUTING (2)

	y_1	y_2
Step 1	$\neg x_2$	-
Step 2	$\neg x_2$	x_2
Step 3	$\neg x_2$	$\neg x_1 \vee x_2$
Step 4	$x_1 \wedge \neg x_2$	$\neg x_1 \vee x_2$

E. Multi-Input Implication

Computation using the architecture shown in Fig. 4 is naturally sequential, because, in many cases, operations cannot be performed in parallel. For instance, it is not possible to compute $x_1 \rightarrow y_1$ in parallel to $x_2 \rightarrow y_2$. Consider an attempt to calculate these operations simultaneously. In this case, *Vcond* is applied to both X_1 and X_2, while *Vset* is applied to both Y_1 and Y_2. Consequently, $x_1 \rightarrow y_2$ and $x_2 \rightarrow y_1$ are also computed. Thus, $x_1(x_2)$ also impacts the final value of $y_2(y_1)$.

Some parallelism can be obtained using multi-input implication [37], [38]. This operations is performed by applying *Vcond* to input memristors $\mathrm{X}_{k0}, \mathrm{X}_{k1}, \ldots, \mathrm{X}_{km}$, $m \leq n$, while *Vset* is applied to a work memristor $\mathrm{Y_j}$. A multi-input implication can be written as

$$(x_{k0} \vee x_{k1} \vee \cdots \vee x_{km}) \rightarrow y_j, \quad j \in \{1, 2\}. \tag{3}$$

An interesting characteristic of multi-input implications is that the number of disjunction operations has no impact on the computation time, i.e., (3) is always computed in a single step.

F. Positive Product Term

Let $S(X)$ be the power set of X. For each $S_k \in S(X), 1 \leq k \leq 2^n$, a positive product term (PPT) π_k, is a conjunction of variables [40]

$$\pi_k = \begin{cases} 1, S_k = \{\} \\ \prod\limits_{x_i \in S_k} x_i, \forall S_k \in S\left(X\right) \setminus \{\} \end{cases}. \tag{4}$$

There are $(2^n)!$ possible orders for S and, consequently, of PPT. We order PPT for $n = 3$ and 4 as depicted in Tables III and IV, respectively.

TABLE III
POSITIVE PRODUCT TERMS FOR $n = 3$

x_1	x_2	x_3	π_k
0	0	0	$\pi_8 = 1$
0	0	1	$\pi_7 = x_3$
0	1	0	$\pi_6 = x_2$
0	1	1	$\pi_4 = x_2 \wedge x_3$
1	0	0	$\pi_5 = x_1$
1	0	1	$\pi_3 = x_1 \wedge x_3$
1	1	0	$\pi_2 = x_1 \wedge x_2$
1	1	1	$\pi_1 = x_1 \wedge x_2 \wedge x_3$

TABLE IV
POSITIVE PRODUCT TERMS FOR $n = 4$

x_1	x_2	x_3	x_4	π_k
0	0	0	0	$\pi_{16} = 1$
0	0	0	1	$\pi_{15} = x_4$
0	0	1	0	$\pi_{14} = x_3$
0	0	1	1	$\pi_{11} = x_3 \wedge x_4$
0	1	0	0	$\pi_{13} = x_2$
0	1	0	1	$\pi_{10} = x_2 \wedge x_4$
0	1	1	0	$\pi_9 = x_2 \wedge x_3$
0	1	1	1	$\pi_5 = x_2 \wedge x_3 \wedge x_4$
1	0	0	0	$\pi_{12} = x_1$
1	0	0	1	$\pi_8 = x_1 \wedge x_4$
1	0	1	0	$\pi_7 = x_1 \wedge x_3$
1	0	1	1	$\pi_4 = x_1 \wedge x_3 \wedge x_4$
1	1	0	0	$\pi_6 = x_1 \wedge x_2$
1	1	0	1	$\pi_3 = x_1 \wedge x_2 \wedge x_4$
1	1	1	0	$\pi_2 = x_1 \wedge x_2 \wedge x_3$
1	1	1	1	$\pi_1 = x_1 \wedge x_2 \wedge x_3 \wedge x_4$

IMPLY expressions are usually written using PPT instead of input variables. If the input memristors are programmed to store the complemented value of input variables (i.e., X_i stores $\neg x_i$), a multi-input implication calculates the complemented value of a PPT ($\neg\pi_k$). In this sense, a multi-input implication takes the form

$$(\neg x_{k1} \vee \neg x_{k2} \vee \cdots \vee \neg x_{km}) \rightarrow y_j, \quad j \in \{1, 2\}. \tag{5}$$

Since $x_{k1} \wedge x_{k2} \wedge \cdots \wedge x_{km}$ equals a π_k, (5) can be written as

$$\neg\pi_k \rightarrow y_j. \tag{6}$$

G. Recursive Forms

The number of work memristors to compute an IMPLY expression is usually a concern in memristive IMPLY stateful logic. In [35], it was shown that recursive forms are always computable with two work memristors. A recursive form is given by

$$f = \left(\neg\pi_1 \rightarrow \left(\neg\pi_2 \rightarrow \cdots \rightarrow \left(\neg\pi_{L-1} \rightarrow {\pi_L}^{\alpha_L}\right)^{\alpha_{L-1}} \cdots\right)^{\alpha_2}\right)^{\alpha_1} \tag{7}$$

where $L = 2^n$ and each α_k is given by

$$\alpha_k = \begin{cases} F(\beta_1), & k = 1 \\ \neg(F(\beta_k) \otimes F(\beta_{k-1})), & k \in [2, L] \end{cases} \tag{8}$$

where $\otimes$ denotes the exclusive disjunction operation and each β_k is a binary n-tuple $\beta_k = \{b_{k1}, b_{k2}, \ldots, b_{kn}\}$ satisfying

$$b_{ki} = 1 \Leftrightarrow x_i \in \pi_k. \tag{9}$$

Equation (7) is valid for any Boolean function *iff* the ordering chosen for the PPT satisfies a partial order [40]

$$(\forall j, k \in \{1, \ldots, L\})(\pi_j \subset \pi_k \rightarrow j > k). \tag{10}$$

In other words, if two PPT share variables, these terms must be ordered according to the number of variables present in the terms. Apart from this constraint, any order can be chosen.

The main drawbacks regarding (7) are that all PPT appear in the equation and have a fixed ordering. Removing these constraints, a recursive form is given by

$$f = \begin{cases} 0 & \text{(11a)} \\ \neg\pi_k \rightarrow f1 & \text{(11b)} \\ f1 \rightarrow 0 & \text{(11c)} \end{cases}$$

where $f1$ is given by (11). Notice that, in (11), a $\neg\pi_k$ is not itself a recursive form. This restriction is useful to force the computation of a $\neg\pi_k$ using two IMPLY operations, storing the resulting value into a work memristor. Even though a $\neg\pi_k$ could be considered a recursive form, in memristive IMPLY logic, the use of a $\neg\pi_k$ at the right side of an IMPLY operation can overwrite input values, as discussed in Section IV.

Intuitively, in a recursive form, the result of the last operation is always used in the succeeding operation. Moreover, only one expression ($\neg\pi_k \rightarrow 0$) can appear in a recursive form.

1) Example 2: Consider a Boolean function F given by

$$f^* = x_1 \wedge x_2 \vee \neg x_3. \tag{12}$$

A recursive IMPLY expression for F is

$$f = \neg\pi_2 \rightarrow ((\neg\pi_7 \rightarrow 0) \rightarrow 0) \tag{13}$$

which is in the form of (11b), being

$$\neg\pi_k = \neg\pi_2 \tag{14a}$$
$$f_1 = (\neg\pi_7 \rightarrow 0) \rightarrow 0. \tag{14b}$$

IV. CORRECTION OF BOOLEAN EXPRESSIONS

Different methods to minimize IMPLY expressions have been presented [39], [40]. In [39], it is proposed an iterative algorithm similar to the Quine-McCluskey method [45].

At each step, the algorithm adds a PPT to the expression aiming to cover a target function. The selection of the PPT uses a greedy strategy, which often leads to suboptimal results. The approach presented in [40], in turn, creates a directed graph in which each vertex is a PPT. There is an edge $(\pi_j, \pi_k)\ iff\ \pi_j \supset \pi_k$. Each vertex is colored accordingly to the Boolean function value for the minterm related to this vertex (e.g., mintern $x_1 \wedge \neg x_2 \wedge \neg x_3$ is associated to PPT $\pi_5 = x_1$). The graph is partitioned in such a way that all vertexes in a given partition are connected and have the same color. In each partition, there is at least one leaf node (vertices without outgoing edges). Only such nodes are used in the final IMPLY expression.

Both algorithms presented in [39] and in [40] consider that reducing the following expression:

$$\neg\pi_a \rightarrow ((\neg\pi_b \rightarrow 0) \rightarrow 0) \tag{15}$$

to the logically equivalent expression

$$\neg\pi_a \rightarrow \neg\pi_b \tag{16}$$

improves the solution quality because (16) is written with two less operations than (15). However, an attempt to compute (16) directly implies that input memristors are used at the right side of the IMPLY operation, leading to loss of input values. Hence, (15) may be considered as the correct solution. Hereafter, expressions containing (16) are named *unsafe*. Otherwise, they are *safe* expressions.

An unsafe expression can always be converted to a safe expression by adding two IMPLY operations, for each subexpression in the form of (16). However, the number of operations can increase more than the necessary. For instance, the 2-input exclusive disjunction (XOR2) has optimal unsafe form given by

$$(\neg\pi_3 \rightarrow \neg\pi_2) \rightarrow ((\neg\pi_2 \rightarrow \neg\pi_3) \rightarrow 0) \tag{17}$$

where the PPT for $n = 2$ are $\pi_1 = x_1 \wedge x_2$, $\pi_2 = x_1$, $\pi_3 = x_2$, and $\pi_4 = 1$. The direct correction yields an expression with eight operators, as follows:

$$(\neg\pi_3 \rightarrow ((\neg\pi_2 \rightarrow 0) \rightarrow 0)) \rightarrow ((\neg\pi_2 \rightarrow ((\neg\pi_3 \rightarrow 0) \rightarrow 0)) \rightarrow 0). \tag{18}$$

However, a safe form for the XOR2 function can be obtained with five operators, as follows:

$$(\neg\pi_1 \rightarrow ((\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0)) \rightarrow 0. \tag{19}$$

Comparing (18) and (19), the trivial correction leads to an overhead of three IMPLY operations. A better evaluation of the impact of such a correction is presented in Section VIII.

V. Multi-memristor Implication

Multi-memristor implication is a generalization of multi-input implication, which allows the use of a work memristor in a multi-input implication operation. The basic principle is to apply *Vcond* to a work memristor and to a set of input memristors, while *Vset* is applied to the other work memristor. This generalization enables operations in the following form to be computed in a single step:

$$(\neg x_{k1} \vee \cdots \vee \neg x_{km} \vee y_i) \rightarrow y_j, \quad i, j \in \{1, 2\}, i \neq j. \tag{20}$$

Alternatively, (21) can be rewritten as follows:

$$(\neg\pi_k \vee y_i) \rightarrow y_j. \tag{21}$$

In (20) and (21) the states of work memristors are explicit. However, memristors states are usually implicit in equations. Thus, a multi-memristor implication operation appears as

$$(\neg\pi_k \vee f1) \rightarrow f2 \tag{22}$$

where $f1$ and $f2$ are IMPLY expressions. If $f1$ is the logic 0, (22) becomes a multi-input implication operation. Hence, expressions using only multi-input implication do not explicitly contain an auxiliary operator '$\vee$'. Given a Boolean expression, the utilization of multi-memristor implication can be identified by the existence of a sub-expression in the form of (22). The definition of recursive form is generalized to include multi-memristor implication operations. Hereafter, a recursive form is given by

$$f = \begin{cases} 0 & \text{(23a)} \\ \neg\pi_k \rightarrow f1 & \text{(23b)} \\ (\neg\pi_k \vee f1) \rightarrow 0 & \text{(23c)} \end{cases}$$

where $f1$ is in the form of (23). Notice that (23c) becomes (11c) when $\neg\pi_k = 0$. Also, (23c) corresponds to (22) with $f2 = 0$. Allowing $f2 \neq 0$ may lead to forms that are not recursive. Such forms are considered in Section VI.

Multi-memristor implication reduces the number of IMPLY operations for any Boolean function F that can be expressed in the following form:

$$f^*(X) = fp^*(Xp) \wedge fn^*(Xn), \quad Xp \cap Xn = \{\}, Xp \cup Xn = X, Xp \neq X, Xn \neq X \tag{24}$$

where both fp^* and fn^* are conjunctions of literals given by

$$fp^* = \prod_{x_p \in Xp} x_p = x_{p1} \wedge x_{p2} \wedge \cdots \wedge x_{p\rho} \tag{25}$$

and

$$fn^* = \prod_{x_n \in Xn} \neg x_n = \neg x_{n1} \wedge \neg x_{n2} \wedge \cdots \wedge \neg x_{n\eta} \tag{26}$$

where ρ and η are the number of elements of Xp and Xn, respectively. Since any conjunction of positive literals corresponds to a PPT, fp^* equals a PPT π_{fp}. In turn, fn^* can be written as function of PPT as $fn^* = \neg\pi_{n1} \wedge \neg\pi_{n2} \wedge \cdots \wedge \neg\pi_{n\eta}$, where $\pi_{n1} = x_{n1}, \pi_{n2} = x_{n2}, \ldots, \pi_{n\eta} = x_{n\eta}$. Notice that conjunctions of complemented literals are not PPT. To obtain a multi-input implication expression for F, f^* is rewritten as follows:

$$f^* = \neg(\neg\pi_{fp} \vee \pi_{n1} \vee \pi_{n2} \vee \cdots \vee \pi_{n\eta}). \tag{27}$$

An IMPLY expression is obtained as follows:

$$f = (\neg\pi_{n\eta} \rightarrow \cdots \rightarrow (\neg\pi_{n1} \rightarrow ((\neg\pi_{fp} \rightarrow 0) \rightarrow 0)) \ldots) \rightarrow 0 \tag{28}$$

which requires $\eta + 3$ operations. Each PPT in fn^* adds an IMPLY operation. The term π_{fp} contributes with two implications and the final complementation operation adds another implication. The number of operations does not depend on ρ because any PPT is computed in a single step. If multi-memristor implication is used, f becomes

$$f = (\neg\pi_{fp} \vee (\neg\pi_{n\eta} \rightarrow \cdots \rightarrow (\neg\pi_{n1} \rightarrow 0)\ldots)) \rightarrow 0 \quad (29)$$

which requires $\eta + 1$ operations. Thus, two less operations are needed when multi-memristor implication is applied. Expression (29) is in the form of (23c) being

$$f1 = \neg\pi_{n\eta} \rightarrow \cdots \rightarrow (\neg\pi_{n2} \rightarrow (\neg\pi_{n1} \rightarrow 0)). \quad (30)$$

1) Example 3: As example of the application of multi-memristor implication, consider the synthesis of a function described by

$$f^* = x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4. \quad (31)$$

Expression (31) is in the form of (24), with $fp^* = x_1 \wedge x_2 = \pi_6$ and $fn^* = \neg x_3 \wedge \neg x_4 = \neg\pi_{14} \wedge \neg\pi_{15}$. From (28), (31) is implemented using multi-input implication as

$$f = (\neg\pi_{15} \rightarrow (\neg\pi_{14} \rightarrow ((\neg\pi_6 \rightarrow 0) \rightarrow 0))) \rightarrow 0. \quad (32)$$

Expression (32) requires five IMPLY operations. A Boolean expression using the proposed multi-memristor implication can be found from (29), as follows:

$$f = (\neg\pi_6 \vee (\neg\pi_{15} \rightarrow (\neg\pi_{14} \rightarrow 0))) \rightarrow 0 \quad (33)$$

which can be computed with three IMPLY operations. The computation of (33) can be performed as follows.

Step 1: $y_1 = 0$. $\neg\pi_{14} \rightarrow y_1$.
Step 2: $\neg\pi_{15} \rightarrow y_1$.
Step 3: $y_2 = 0$. $(\neg\pi_6 \vee y_1) \rightarrow y_2$. To perform the last operation, apply *Vcond* to Y_1 and to all input memristors in $\neg\pi_6$ (X_1 and X_2), while applying *Vset* to Y_2, which will keep the value of f. Since both input memristors and a work memristor appear at the left side of IMPLY operation, this step performs a multi-memristor implication.

VI. Factored Forms

Existing algorithms for reducing the number of IMPLY operations in Boolean expressions guarantee that the resulting expression is computable with two work memristors [39], [40] . These methods consider only recursive forms because any recursive form can be computed with two work memristors [35].

A contribution of this work is the utilization of factored forms to reduce the number of IMPLY operations. A material implication based factored form can be defined as

$$f = \begin{cases} 0 & (34a) \\ \neg\pi_k \rightarrow f1 & (34b) \\ (\neg\pi_k \vee f1) \rightarrow f2 & (34c) \\ f1 \rightarrow f2 & (34d) \end{cases}$$

where both $f1$ and $f2$ are given by (34). Notice that (34b) and (34c) are reduced to (23b) and (23c), respectively, when $f2 = 0$. Hence, any recursive form is a factored form though not all factored form is a recursive one. Since factored forms are more general than recursive ones, it may be possible to obtain smaller IMPLY expressions. However, not all factored forms can be computed with two work memristors. Therefore, the number of work memristors necessary to compute a Boolean function becomes a concern.

In the following, we discuss which factored forms can be computed with two work memristors. Moreover, a simple method to determine the number of work memristor necessary to compute an IMPLY expression is presented. We use $\mu(f)$ to denote the number of work memristors required to compute an IMPLY expression f.

The number of work memristors to compute (34b) equals to the maximum between one and $\mu(f1)$. Expressions given by (34c) or (34d) can be computed using two work memristors if $\mu(f1)(\mu(f2))$ is smaller or equal than two and $\mu(f2)(\mu(f1))$ is at most one (i.e., $\mu(f1) + \mu(f2) \leq 3$). In this case, the function requiring more work memristors must be computed first. The reasoning for this argument is as follows. Let $\mu(f1) = 1$ and $\mu(f2) = 2$. Compute $f2$ using both work memristors available. Once $f2$ is evaluated, one work memristor must store the resulting value. However, the other work memristor, can be used to calculate $f1$. After $f1$ is computed, each work memristor stores either $f1$ or $f2$ and the last IMPLY can be performed. On the other hand, if $f1$ is evaluated before $f2$, a third work memristor must be added because there is only one work memristor available to compute $f2$. If $\mu(f1) = \mu(f2) = 2$, three work memristors are required regardless of the order chosen to compute $f1$ and $f2$.

The minimal number of work memristors to compute a factored form given by (34c) or (34d) is determined from the number of work memristors to compute $f1$ and $f2$ as follows:

$$\mu(f) = \begin{cases} \max(\mu(f1), \mu(f2)) & \mu(f1) \neq \mu(f2) \\ \mu(f2) + 1 & \mu(f1) = \mu(f2) \end{cases}. \quad (35)$$

If $\mu(f1)$ and $\mu(f2)$ are different, the greater value between them is chosen. In this case, the function that requires more work memristors must be calculated before the other one. When $\mu(f1)$ and $\mu(f2)$ are equal, a work memristor must be added, regardless the ordering chosen to compute $f1$ and $f2$.

1) Example 4: Consider the computation of a Boolean function F represented by

$$f = (\neg\pi_b \rightarrow ((\neg\pi_a \rightarrow 0) \rightarrow 0)) \rightarrow (\neg\pi_d \rightarrow ((\neg\pi_c \rightarrow 0)) \quad (36)$$

where π_a, π_b, π_c, and π_d are arbitrary PPT. Expression (36) is in the form of (34c), being

$$f1 = (\neg\pi_b \rightarrow ((\neg\pi_a \rightarrow 0) \rightarrow 0)) \quad (37a)$$
$$f2 = (\neg\pi_d \rightarrow ((\neg\pi_c \rightarrow 0) \rightarrow 0)). \quad (37b)$$

The last IMPLY operation $(f1 \rightarrow f2)$ can only be performed after the computation of (37a) and (37b). The steps required to compute f are shown in the following. Notice the need to

TABLE V
STATES OF WORK MEMRISTORS AT EACH STEP WHEN COMPUTING (36)

	y_1	y_2	y_3
Step 1	π_a	-	-
Step 2	π_a	$\neg\pi_a$	-
Step 3	π_a	$\pi_b \vee \neg\pi_a$	-
Step 4	π_c	$\pi_b \vee \neg\pi_a$	-
Step 5	π_c	$\pi_b \vee \neg\pi_a$	$\neg\pi_c$
Step 6	π_c	$\pi_b \vee \neg\pi_a$	$\pi_d \vee \neg\pi_c$
Step 7	π_c	$\pi_b \vee \neg\pi_a$	$\neg\pi_b \wedge\pi_a \vee \pi_d \vee \neg\pi_c$

TABLE VI
STATES OF WORK MEMRISTORS AT EACH STEP WHEN COMPUTING (38)

	y_1	y_2
Step 1	π_a	-
Step 2	π_a	$\neg\pi_a$
Step 3	π_a	$\pi_b \vee \neg\pi_a$
Step 4	π_c	$\pi_b \vee \neg\pi_a$
Step 5	$\pi_d \vee \pi_c$	$\pi_b \vee \neg\pi_a$
Step 6	$\neg\pi_b \wedge \pi_a \vee \pi_d \vee \pi_c$	$\pi_b \vee \neg\pi_a$

include a third work memristor in Step 5 (similar analysis holds when $f1$ is computed before $f2$).

Step 1: $y_1 = 0, \neg\pi_a \rightarrow y_1$. Y_1 stores π_a.

Step 2: $y_2 = 0, y_1 \rightarrow y_2$. Y_2 stores $\neg\pi_a$.

Step 3: $\neg\pi_b \rightarrow y_2$. Y_2 stores $\pi_b \vee \neg\pi_a$. This step concludes the computation of $f1$.

Step 4: $y_1 = 0, \neg\pi_c \rightarrow y_1$. Y_1 stores π_c. Since Step 4 does not use the result from Step 3, (36) is not a recursive form.

Step 5: $y_3 = 0, y_1 \rightarrow y_3$. In this step, a complementation operation on y_1 must be performed. However, Y_2 stores the value of $f1$ and cannot be used to keep the resulting value. The solution is to add a work memristor Y_3.

Step 6: $\neg\pi_d \rightarrow y_3$. This step concludes the computation of $f2$, which is stored on Y_3.

Step 7: $y_2 \rightarrow y_3$. This step performs the last IMPLY $(f1 \rightarrow f2)$. The final result is stored on Y_3.

The states of work memristors at the end of each step are shown in Table V. A '$-$' sign indicates that the state of the work memristor is not important at that stage. Notice that both $f1$ and $f2$ require two work memristors. Expression $f1$ uses Y_1 and Y_2 whereas $f2$ uses Y_1 and Y_3. As expected from (35), three work memristors are required to compute f.

2) Example 5: An example of factored form computable with two work memristors is the following:

$$f = f1 \rightarrow f3 \tag{38}$$

where $f1$ is given by (37a) and $f3$ is given by

$$f3 = (\neg\pi_d \rightarrow (\neg\pi_c \rightarrow 0)). \tag{39}$$

To compute (38), steps 1 to 4 are the same from example 4. The remaining steps are the following.

Step 5: $\neg\pi_d \rightarrow y_1$. This step concludes the computation of $f3$, which is stored in Y_1. In contrast to Example 4, no work memristor must be added at this step.

Step 6: $y_2 \rightarrow y_1$. This step performs the last IMPLY $(f1 \rightarrow f3)$. Y_1 keeps the final result.

Since $\mu(f1) = 2$ and $\mu(f3) = 1$, two work memristors suffice to compute (38). Table VI presents the state of work memristors at the end of each step.

3) Example 6: The optimal equation for the XOR2 function, computable with two work memristors, uses both a factored form and multi-memristor implication:

$$(\neg\pi_2 \vee (\neg\pi_3 \rightarrow 0)) \rightarrow ((\neg\pi_3 \vee (\neg\pi_2 \rightarrow 0)) \rightarrow 0). \tag{40}$$

Equation (40) is in the form of (34d), being

$$f1 = \neg\pi_3 \rightarrow 0 \tag{41a}$$

$$f2 = (\neg\pi_3 \vee (\neg\pi_2 \rightarrow 0)) \rightarrow 0. \tag{41b}$$

Since $\mu(f1) = 1$ and $\mu(f2) = 2$, (40) can be computed with two work memristors. Equation (40) is evaluated with one less IMPLY operation than the recursive form given by (19).

VII. SYNTHESIS OF IMPLY EXPRESSIONS

This section presents an algorithm to synthesize IMPLY expressions with minimized number of operations. The main contribution of the proposed method in comparison to previous works [39], [40] is the possibility to take into account different expression forms, improving the solution quality. We propose a bottom-up approach to perform synthesis of Boolean expressions, based on the factoring algorithm presented in [46]. The method obtains expressions (also named implementations) for complex functions by combining simpler expressions.

The algorithm proposed herein aims to reduce the number of IMPLY operations, so an optimal implementation comprises the minimum number of such operators. Hereafter, the term c-cost function is adopted to refer to a function with implementation cost c, i.e., the implementation of the target function requires c IMPLY operations. The set of c-cost functions is denoted by C_c. An implementation with cost c is denoted by f_c.

The algorithm starts from the set of 0-cost functions, which are all complemented PPT $(\neg\pi_k)$, including the constant value 0. This choice is made because Boolean expressions are written in terms of $\neg\pi_k$. Consequently, PPT comprise 1-cost functions because one IMPLY is needed to implement these functions $(\neg\pi_k \rightarrow 0)$. Notice that the constant 1 is not used since $(1 \rightarrow f = f)$ and $(f \rightarrow 1 = 1)$.

When implementations are combined, each resulting expression has a known cost. Keeping the optimal implementation of each function, the best solutions for more complex functions are also obtained. Notice that if a function f has optimal implementations as $f = g \rightarrow h$, implementations g and h must also be optimal. When all possible c-cost functions are generated, the algorithm proceeds to generate the set of $(c+1)$-cost functions, until a solution is found. Therefore, the first implementation found for a given function is also optimal. We assume that the resulting value of an expression is always stored in a work

TABLE VII
RECURSIVE MULTI-INPUT IMPLICATION EXPRESSIONS FOR NON-CONSTANT BOOLEAN FUNCTIONS UP TO TWO INPUTS, CLASSIFIED BY COST

C_c		Implementations
C_1		$\neg\pi_1 \rightarrow 0,\ \neg\pi_2 \rightarrow 0,\ \neg\pi_3 \rightarrow 0$
C_2	C_{2a}	$\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)$
	C_{2b}	$(\neg\pi_1 \rightarrow 0) \rightarrow 0,\ (\neg\pi_2 \rightarrow 0) \rightarrow 0,\ (\neg\pi_3 \rightarrow 0) \rightarrow 0$
C_3	C_{3a}	$\neg\pi_2 \rightarrow ((\neg\pi_3 \rightarrow 0) \rightarrow 0),\ \neg\pi_3 \rightarrow ((\neg\pi_2 \rightarrow 0) \rightarrow 0)$
	C_{3b}	$(\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0$
C_4	C_{4a}	$\neg\pi_1 \rightarrow ((\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0)$
	C_{4b}	$(\neg\pi_2 \rightarrow ((\neg\pi_3 \rightarrow 0) \rightarrow 0)) \rightarrow 0,$ $(\neg\pi_3 \rightarrow ((\neg\pi_2 \rightarrow 0) \rightarrow 0)) \rightarrow 0$
C_5		$(\neg\pi_1 \rightarrow ((\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0)) \rightarrow 0$

memristor. Hence, the implementation cost for a non-constant 0-cost function equals two $((\neg\pi_k \rightarrow 0) \rightarrow 0)$. In the following we consider the synthesis of c-cost functions with $c \geq 2$.

A. Recursive Multi-Input Implication Synthesis

Rules to combine expressions, considering only multi-input implication, are in the form of (11). For a given value of c, the possible functions are restricted to those leading to the correct cost. Expressions for a c-cost function can be generated as

$$f_c = \begin{cases} f_0 \rightarrow f_{c-1} & \text{(42a)} \\ f_{c-1} \rightarrow 0. & \text{(42b)} \end{cases}$$

Hence, if f_{c-1} is a recursive form, applying (42a) or (42b) to f_{c-1} creates an expression f_c which is also recursive. Since f_1 functions are recursive, (42) generates recursive expressions.

1) Example 7: Consider the synthesis of all nonconstant Boolean functions with at most 2-inputs.

The sets of basic functions C_0 and C_1 are $\{\neg\pi_1,\ \neg\pi_2,\ \neg\pi_3,\ 0\}$, and $\{\neg\pi_1 \rightarrow 0,\ \neg\pi_2 \rightarrow 0, \neg\pi_3 \rightarrow 0\}$, respectively. These sets contain implementations to 3 of the 14 target functions. Expressions $\neg\pi_1, \neg\pi_2$, and $\neg\pi_3$ in C_0, are not considered valid implementations. Since there are still functions without known solution, C_2 must be computed. Rules to obtain 2-cost functions are attained by replacing $c = 2$ in (42), as follows (rules for higher costs are obtained in similar manner):

$$f_2 = \begin{cases} f_0 \rightarrow f_1 & \text{(43a)} \\ f_1 \rightarrow 0. & \text{(43b)} \end{cases}$$

When generating the set of f_2 functions, all combinations of f_1 and f_0 functions are applied. However, the resulting expression is only stored if it is the first implementation for the resulting function.

The implementations for all nonconstant Boolean functions with at most two inputs are shown in Table VII, where each set C_c is given in a line. The subsets C_{ca} and C_{cb} refer to equations in the form of (42a) and (42b), respectively.

The same approach can be used to find the implementation of a single function. In this case, functions sets are created until the target function is found.

TABLE VIII
RECURSIVE MULTI-MEMRISTOR IMPLICATION EXPRESSIONS FOR NONCONSTANT BOOLEAN FUNCTIONS UP TO TWO INPUTS, CLASSIFIED BY COST

C_c		Implementations
C_1		$\neg\pi_1 \rightarrow 0,\ \neg\pi_2 \rightarrow 0,\ \neg\pi_3 \rightarrow 0$
C_2	C_{2a}	$\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)$
	C_{2b}	$(\neg\pi_1 \rightarrow 0) \rightarrow 0, (\neg\pi_2 \rightarrow 0) \rightarrow 0, (\neg\pi_3 \rightarrow 0) \rightarrow 0,$ $(\neg\pi_3 \vee (\neg\pi_2 \rightarrow 0)) \rightarrow 0,\ (\neg\pi_2 \vee (\neg\pi_3 \rightarrow 0)) \rightarrow 0$
C_3	C_{3a}	$\neg\pi_2 \rightarrow ((\neg\pi_3 \rightarrow 0) \rightarrow 0),\ \neg\pi_3 \rightarrow ((\neg\pi_2 \rightarrow 0) \rightarrow 0)$
	C_{3b}	$(\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0$
C_4		$\neg\pi_1 \rightarrow ((\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0)$
C_5		$(\neg\pi_1 \rightarrow ((\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow 0)) \rightarrow 0)) \rightarrow 0$

2) Example 8: Consider a function F described by

$$f^* = \neg x_1 \vee x_2. \tag{44}$$

After C_2 is created, the algorithm checks if an implementation for F exists. Since no expression in C_2 represents (44), set C_3 must be calculated. An implementation for (44) is found in set C_3, as follows:

$$f_3 = \neg\pi_3 \rightarrow ((\neg\pi_2 \rightarrow 0) \rightarrow 0). \tag{45}$$

Hence the algorithm stops without creating sets C_4 and C_5. Equation (45) is obtained by applying (42b) to the 1-cost function $(\neg\pi_2 \rightarrow 0)$, followed by (42a), with $f_0 = \neg\pi_3$, to the resulting expression.

B. Recursive Multi-Memristor Implication Synthesis

To synthesize recursive forms using multi-memristor implication, (42) is generalized to obtain expressions in the form of (23). A c-cost function f_c can be obtained by

$$f_c = \begin{cases} f_0 \rightarrow f_{c-1} & \text{(46a)} \\ (f_{c-1} \vee f_0) \rightarrow 0. & \text{(46b)} \end{cases}$$

Notice that (46b) is reduced to (42b), when $f_0 = 0$. Such behavior is consistent with the fact that multi-memristor implication is a generalization of multi-input implication.

1) Example 9: Consider the synthesis of all Boolean functions with at most two inputs. Optimal implementations are given in Table VIII. Sets C_0 and C_1 are the same as in Example 7. Subsets C_{ca} and C_{cb} contain c-cost functions in the form of (46a) and (46b), respectively.

Comparing Tables VII and VIII, the implementations of two functions are improved. These functions are represented by

$$f1^* = \neg x_1 \wedge x_2 \tag{47a}$$
$$f2^* = x_1 \wedge \neg x_2. \tag{47b}$$

Such a reduction is expected since both (47a) and (47b) are in the form of (24). The expressions for (47a) obtained from Tables VII and VIII are, respectively, $f1_a$ and $f1_b$ [similar expressions are obtained for (47b)]

$$f1_a = (\neg\pi_2 \rightarrow ((\neg\pi_3 \rightarrow 0) \rightarrow 0)) \rightarrow 0 \tag{48a}$$
$$f1_b = (\neg\pi_3 \vee (\neg\pi_2 \rightarrow 0)) \rightarrow 0. \tag{48b}$$

C. Factored Multi-Input Implication Synthesis

Synthesis of factored forms should consider the number of work memristors required to compute the resulting expression. Factored forms, computable with 2 work memristors, using only multi-input implication, can be obtained from (34). However, the possibility to perform a multi-memristor operation is removed from (34c). The resulting set of equations is given as follows:

$$f_c = \begin{cases} f_0 \rightarrow f_{c-1} & \text{(49a)} \\ f_{c-1} \rightarrow 0 & \text{(49b)} \\ f_{c-a} \rightarrow f_{a-1}, \mu(f_{c-a}) + \mu(f_{a-1}) \leq 3 & \text{(49c)} \end{cases}$$

where a is an integer and $1 < a < c$. The condition in (49c) ensures that resulting expressions are computable with 2 work memristors, as discussed in Section VI. The number of work memristors to compute (49a) or (49b) is determined by f_{c-1}.

1) Example 10: Consider a target function F given by

$$f^* = (\neg x_1 \wedge x_2) \vee (\neg x_3 \wedge \neg x_4). \quad (50)$$

Following the same procedure described in previous examples, a multi-input implication factored form $(f1)$ for F appears in C_7, as follows:

$$f_1 = (\neg\pi_{14} \rightarrow (\neg\pi_{15} \rightarrow 0)) \rightarrow ((\neg\pi_{12} \rightarrow ((\neg\pi_{13} \rightarrow 0) \rightarrow 0)) \rightarrow 0) \quad (51)$$

where $\neg\pi_{12} = \neg x_1$, $\neg\pi_{13} = \neg x_2$, $\neg\pi_{14} = \neg x_3$, $\neg\pi_{15} = \neg x_4$. Equation (51) is in the form of (49c), being

$$f_{c-a} = f_2 = (\neg\pi_{14} \rightarrow (\neg\pi_{15} \rightarrow 0)) \quad (52a)$$
$$f_{a-1} = f_4 = ((\neg\pi_{12} \rightarrow ((\neg\pi_{13} \rightarrow 0) \rightarrow 0)) \rightarrow 0). \quad (52b)$$

Equation (52a) is in the form of (49a) with

$$f_0 = \neg\pi_{14} \quad (53a)$$
$$f_{c-1} = f_1 = \neg\pi_{15} \rightarrow 0. \quad (53b)$$

In turn, (52b) is given by (49b), being

$$f_{c-1} = f_3 = (\neg\pi_{12} \rightarrow ((\neg\pi_{13} \rightarrow 0) \rightarrow 0)). \quad (54)$$

The optimal recursive form $(f2)$ for F has eight operations

$$f2 = (\neg\pi_8 \rightarrow (\neg\pi_7 \rightarrow ((\neg\pi_{13} \rightarrow f2a) \rightarrow 0))) \rightarrow 0 \quad (55a)$$
$$f2a = ((\neg\pi_{15} \rightarrow (\neg\pi_{14} \rightarrow 0)) \rightarrow 0) \quad (55b)$$

where $\neg\pi_7 = \neg(x_1 \wedge x_3)$ and $\neg\pi_8 = \neg(x_1 \wedge x_4)$. Hence the utilization of a factored form, computable with two work memristors, leads to a reduction of one operation.

D. Factored Multi-Memristor Implication Synthesis

Synthesis of factored forms using multi-memristor implications must also consider the required number of work memristors. The possible combinations to synthesize a c-cost function are taken directly from (34), as follows:

$$f_c = \begin{cases} f_0 \rightarrow f_{c-1} & \text{(56a)} \\ (f_{c-1} \vee f_0) \rightarrow 0 & \text{(56b)} \\ (f_{c-a} \vee f_0) \rightarrow f_{a-1}, \mu(f_{c-a}) + \mu(f_{a-1}) \leq 3. & \text{(56c)} \end{cases}$$

One example of the application of factored forms using multi-memristor implication is the XOR2 function [see (40)].

1) Example 11: Consider the synthesis of the following three-input Boolean function:

$$f^* = \neg x_1 \wedge (x_2 \vee x_3). \quad (57)$$

By setting $c = 2$, 3, and 4 in (56), rules to generate f_2, f_3 and f_4 functions are found. For each value of c, all possible values for a, $1 < a < c$, are considered. The resulting set of rules is given by

$$f_2 = \begin{cases} f_0 \rightarrow f_1 & \text{(58a)} \\ (f_1 \vee f_0) \rightarrow 0 & \text{(58b)} \end{cases}$$

$$f_3 = \begin{cases} f_0 \rightarrow f_2 & \text{(59a)} \\ (f_2 \vee f_0) \rightarrow 0 & \text{(59b)} \\ (f_1 \vee f_0) \rightarrow f_1 & \text{(59c)} \end{cases}$$

$$f_4 = \begin{cases} f_0 \rightarrow f_3 & \text{(60a)} \\ (f_3 \vee f_0) \rightarrow 0 & \text{(60b)} \\ (f_2 \vee f_0) \rightarrow f_1 & \text{(60c)} \\ (f_1 \vee f_0) \rightarrow f_2. & \text{(60d)} \end{cases}$$

Notice that when $c = 2$, there is no valid value for a and (56c) is not used. Consequently, there are only two rules in (58). For $c = 3$, a can only assume the value 2. Finally, for $c = 4$, a can be either 2 or 3.

After generating sets C_2, C_3 and C_4, the best implementation for $F(f1)$ is found with 4 material implications, as follows:

$$f_1 = ((\neg\pi_5 \rightarrow 0) \vee \neg\pi_6) \rightarrow ((\neg\pi_7 \vee (\neg\pi_5 \rightarrow 0)) \rightarrow 0). \quad (61)$$

Equation (61) is in the form of (60d), being

$$f_0 = \neg\pi_6 \quad (62a)$$
$$f_{c-a} = f_1 = \neg\pi_5 \rightarrow 0 \quad (62b)$$
$$f_{a-1} = f_2 = (\neg\pi_7 \vee (\neg\pi_5 \rightarrow 0)) \rightarrow 0. \quad (62c)$$

In turn, (62c) is in the form of (58b), being

$$f_{c-1} = f_1 = \neg\pi_5 \rightarrow 0 \quad (63a)$$
$$f_0 = \neg\pi_7. \quad (63b)$$

The best recursive IMPLY expression for F $(f2)$ has five operators

$$f_2 = (\neg\pi_5 \rightarrow ((\neg\pi_6 \rightarrow (\neg\pi_7 \rightarrow 0)) \rightarrow 0)) \rightarrow 0. \quad (64)$$

VIII. Experimental Results

As presented in [39] and in [40], we also consider implementations of all functions with at most four inputs. Moreover, the metric applied is the number of IMPLY operations, neglecting eventual reset operations.

The first comparison takes into account only recursive expressions using multi-input memristor implication and allowing unsafe expressions. This way, a direct comparison to previous

TABLE IX
AVERAGE NUMBER OF MATERIAL IMPLICATION OPERATIONS TAKING INTO ACCOUNT RECURSIVE FORMS USING UNSAFE MULTI-INPUT IMPLICATION EXPRESSIONS

	Average number of implications	Normalized average
This work	8.83	1.00
[40]	8.91	1.01
[39]	12.00	1.35

TABLE X
AVERAGE NUMBER OF MATERIAL IMPLICATION OPERATIONS TO SYNTHESIZE SAFE EXPRESSIONS TAKING INTO ACCOUNT DIFFERENT APPROACHES

	Multi-input implication	Multi-memristor implication
Recursive	9.26	8.85
Factored	9.24	8.14

works presented in [39] and in [40] is possible. Table IX presents the average number of operations obtained from each algorithm. Notice that results presented in Table IX cannot be considered as the actual number of IMPLY operations since corrections are still needed. Hereafter, only the synthesis of safe expressions is discussed.

The second experiment carried out evaluated the overhead resulted from correcting unsafe Boolean expressions. When the trivial correction is applied, the average number of IMPLY operations grows from 8.83 to 9.45 (7% increase). The correction impact on results obtained from [38] is similar to the one observed herein. Therefore, the corrected number of 9.45 can also be considered as the effective average number of operations using the method proposed in [40].

When our method synthesizes safe expressions directly, the average number of operations grows from 8.83 to 9.26. Still, this represents a small reduction of 2% compared to the correction approach (9.45). Hereafter, all Boolean expressions are synthesized direct to safe forms.

The third experiment evaluated both recursive and factored forms using multi-input implication and multi-memristor implication (four combinations). As shown in Table X, the utilization of multi-memristor implication together with factored forms leads to a reduction of 12% in comparison to the traditional recursive forms. Interestingly, if only multi-input implication is adopted, the reduction obtained by applying factored forms is small.

Fig. 5 shows the distribution regarding the number of IMPLY operations to compute four-input Boolean functions, considering different forms. For all approaches the worst case is the four-input exclusive-disjunction (XOR4) function which requires 19 IMPLY operations.

From the 65 534 evaluated functions, the recursive multi-memristor implication reduced the number of operations in 16 456 cases, i.e., around 25% when compared to the conventional recursive multi-input implication. From this set of functions, in 10 120 cases the reduction was of two operations, in 6240 one less operation was required, and in the remaining 96 cases the reduction was of four operations. The utilization

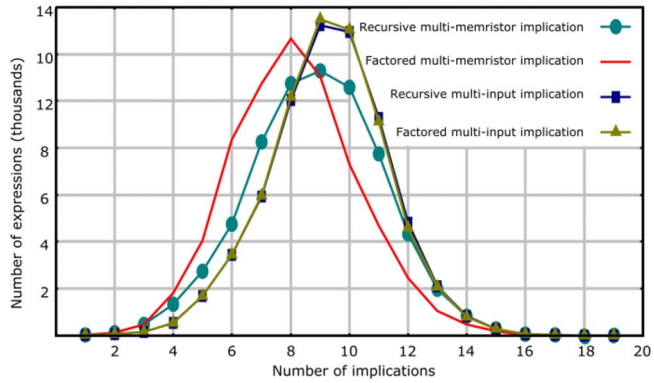

Fig. 5. Number of material implication operations when synthesizing all four variable functions considering different approaches.

TABLE XI
REDUCTION IN THE NUMBER OF OPERATIONS USING MULTI-MEMRISTOR IMPLICATION IN CONJUNCTION WITH RECURSIVE OR FACTORED EXPRESSIONS COMPARED TO THE TRADITIONAL RECURSIVE MULTI-INPUT IMPLICATION OPERATIONS

Reduction	Multi-memristor implication (recursive)	Multi-memristor implication (factored)
0	49,078	21,703
1	6,240	19,745
2	10,120	19,354
3	0	3,981
4	96	664
5	0	63
6	0	24

of factored forms reduced the number of operations of additional 27 375 functions, totalizing 43 831 functions. That is, around 67% of the functions presented better implementations. Table XI shows the gain in the number of implications when multi-memristor implication in both recursive and factored forms is compared to traditional multi-input implication.

IX. CONCLUSION

This paper proposed the utilization of factored forms together with the novel concept of multi-memristor implication in the logic synthesis for memristive IMPLY stateful logic. As a result, the average number of IMPLY operations to perform a function with at most four inputs was reduced by 12%, when compared to previous works, without requiring additional devices. Moreover, implementations of approximately 67% of the Boolean functions considered have been improved.

REFERENCES

[1] L. O. Chua, "Memristor—The missing circuit element," *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep. 1971.
[2] L. O. Chua and S.-M. S. Kang, "Memristive devices and systems," *Proc. IEEE*, vol. 64, no. 2, pp. 209–223, Feb. 1976.
[3] T. Prodromakis, C. Toumazou, and L. Chua, "Two centuries of memristors," *Nature Mater.*, vol. 11, no. 6, pp. 478–481, Jun. 2012.
[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.
[5] C. Toumazou, J. Georgiou, and E. M. Drakakis, "Current-mode analogue circuit representation of Hodgkin and Huxley neuron equations," *Electron. Lett.*, vol. 34, no. 14, pp. 1376–1377, 1998.

[6] A. Ascoli, T. Schmidt, R. Tetzlaff, and F. Corinto, "Application of the volterra series paradigm to memristive systems," in *Memristors and Memristive Syst*, R. Tetzlaff, Ed. New York: Springer, 2014, pp. 163–191.
[7] F. Corinto, A. Ascoli, and M. Gilli, "Nonlinear dynamics of memristors oscillators," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 6, pp. 1323–1336, Jun. 2011.
[8] C. Yakopcic, T. M. Taha, G. Subramanyam, R. E. Pino, and S. Rogers, "A memristor device model," *IEEE Electron Device Lett.*, vol. 32, no. 10, pp. 1436–1438, Oct. 2011.
[9] E. Linn, A. Siemon, R. Waser, and S. Menzel, "Applicability of well-established memristive models for simulations of resistive switching devices," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 8, pp. 2402–2410, Aug. 2014.
[10] B. Mohammad, D. Homouz, and H. Elgabra, "Robust hybrid memristor-CMOS memory: Modeling and design," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 21, no. 11, pp. 2069–2079, Nov. 2013.
[11] Y. Ho, G. M. Huang, and P. Li, "Dynamical properties and design analysis for nonvolatile memristor memories," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 4, pp. 724–736, Apr. 2011.
[12] K. Eshraghian *et al.*, "Memristor MOS content addressable memory (MCAM): Hybrid architecture for future high performance search engines," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 19, no. 8, pp. 1407–1417, Aug. 2011.
[13] E. Lehtonen, J. H. Poikonen, M. Laiho, and P. Kanerva, "Large-scale memristive associative memories," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 22, no. 3, pp. 562–574, Mar. 2014.
[14] H. Kim, M. P. Sah, C. Yang, T. Roska, and L. O. Chua, "Neural synaptic weighting with a pulse-based memristor circuit," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 1, pp. 148–158, Jan. 2012.
[15] Y. V. Pershin and M. Di Ventra, "Neuromorphic, digital, and quantum computation with memory circuit elements," *Proc. IEEE*, vol. 100, no. 6, pp. 2071–2080, Jun. 2012.
[16] T. Serrano-Gotarredona, T. Prodromakis, and A. Linares-Barranco, "A proposal for hybrid memristor-CMOS spiking neuromorphic learning systems," *IEEE Circuits Syst. Mag.*, vol. 13, no. 2, pp. 74–88, 2013.
[17] T. Chang, Y. Yang, and W. Lu, "Building neuromorphic circuits with memristive devices," *IEEE Circuits Syst. Mag.*, vol. 13, no. 2, pp. 56–73, 2013.
[18] J. A. Starzyk and Basawaraj, "Memristor crossbar architecture for synchronous neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 8, pp. 2390–2401, Aug. 2014.
[19] Y. V. Pershin and M. Di Ventra, "Practical approach to programmable analog circuits with memristors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 57, no. 8, pp. 1857–1864, Aug. 2010.
[20] S. Shin, K. Kyungmin, and S.-M. S. Kang, "Memristor applications for programmable analog ICs," *IEEE Trans. Nanotechnol.*, vol. 10, no. 2, pp. 266–274, Mar. 2011.
[21] R. Berdan, T. Prodromakis, I. Salaoru, A. Khiat, and C. Toumazou, "Memristive devices as parameter setting elements in programmable gain amplifiers," *Appl. Physics Lett.*, vol. 101, no. 24, p. 243502, Dec. 2012.
[22] J. Rajendran, H. Manem, R. Karri, and G. S. Rose, "An energy-efficient memristive threshold logic circuit," *IEEE Trans. Comput.*, vol. 61, no. 4, pp. 474–487, Apr. 2012.
[23] A. P. James, L. R. V. J. Francis, and D. S. Kumar, "Resistive threshold logic," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 22, no. 1, pp. 190–195, Jan. 2014.
[24] D. Fan, M. Sharad, and K. Roy, "Design and synthesis of ultralow energy spin-memristor threshold logic," *IEEE Trans. Nanotechnol.*, vol. 13, no. 3, pp. 574–583, May 2014.
[25] L. Gao, F. Alibart, and D. Strukov, "Programmable CMOS/memristor threshold logic," *IEEE Trans. Nanotechnol.*, vol. 12, no. 2, pp. 115–119, Mar. 2013.
[26] Q. F. Xia *et al.*, "Memristor-CMOS hybrid integrated circuits for reconfigurable logic," *Nano Lett.*, vol. 9, no. 10, pp. 3640–3645, 2009.
[27] D. B. Strukov and K. K. Likharev, "CMOL FPGA: a reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices," *Nanotechnology*, vol. 16, no. 6, pp. 888–900, 2005.
[28] G. S. Snider and R. S. Williams, "Nano/CMOS architectures using a field-programmable nanowire interconnect," *Nanotechnology*, vol. 18, no. 3, p. 035204, 2007.
[29] S. Shin, K. Kim, and S.-M. Kang, "Resistive computing: Memristors-enabled signal multiplication," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 60, no. 5, pp. 1241–1249, May 2013.
[30] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, Feb. 2010.
[31] I. Vourkas and G. C. Sirakoulis, "A novel design and modeling paradigm for memristor-based crossbar circuits," *IEEE Trans. Nanotechnol.*, vol. 11, no. 6, pp. 1151–1159, Nov. 2012.
[32] S. Kvatinsky *et al.*, "MAGIC—Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.
[33] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, "Beyond von neumann-logic operations in passive crossbararrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, Aug. 2012.
[34] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE J. Emerg. Select. Topics Circuits Syst.*, vol. 5, no. 1, pp. 64–74, Mar. 2015.
[35] E. Lehtonen, J. H. Poikonen, and M. Laiho, "Two memristors suffice to compute all Boolean functions," *Electron. Lett.*, vol. 46, no. 3, pp. 239–240, Feb. 2010.
[36] K. Kim, S. Shin, and S.-M. S. Kang, "Field programmable stateful logic array," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 30, no. 12, pp. 1800–1813, Dec. 2011.
[37] M. Laiho and E. Lehtonen, "Cellular nanoscale network cell with memristors for local implication logic and synapses," in *Proc. Int. Symp. Circuits Syst.*, 2010, pp. 2051–2054.
[38] S. Shin, K. Kim, and S.-M. S. Kang, "Reconfigurable stateful NOR gate for large-scale logic-array integrations," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 58, no. 7, pp. 442–446, Jul. 2011.
[39] J. H. Poikonen, E. Lehtonen, and M. Laiho, "On synthesis of Boolean expressions for memristive devices using sequential implication logic," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 31, no. 7, pp. 1129–1134, Jul. 2012.
[40] P. Teodorovic, S. Dautovic, and V. Malbasa, "Recursive Boolean formula minimization algorithms for implication logic," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 32, no. 11, pp. 1829–1833, Nov. 2013.
[41] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnol.*, vol. 8, no. 1, pp. 13–24, 2013.
[42] S. Kvatinsky, E. Friedman, A. Kolodny, and U. Weiser, "The desired memristor for circuit designers," *IEEE Circuits Syst. Mag.*, vol. 13, no. 2, pp. 17–22, 2013.
[43] S. Kvatinsky *et al.*, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.
[44] X. Zhu *et al.*, "Performing stateful logic on memristor memory," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 60, no. 10, pp. 682–686, Oct. 2013.
[45] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol. 35, no. 5, pp. 1417–1444, 1956.
[46] M. G. A. Martins, R. P. Ribas, and A. I. Reis, "Functional composition: A new paradigm for performing logic synthesis," in *Proc. Int. Symp. Quality Electron. Design*, 2012, pp. 236–242.

Felipe S. Marranghello (S'14) received the B.S. degree in computer engineering in 2009 and the M.S. degree in microelectronics in 2012, both from Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, where he is currently pursuing the Doctor degree in microelectronics.

His research interests include timing analysis of digital VLSI circuits and logic synthesis techniques for CMOS and emerging technologies.

Vinicius Callegaro (S'14) received the B.S. and M.S. degrees in computer science, in 2010 and 2012, respectively, from Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, where he is currently pursuing the Doctor degree in computer science.

His research interests include logic synthesis methods for CMOS and emerging technologies, as well as automatic generation of standard cell libraries.

Mr. Callegaro received a best paper award at SBCCI'13 conference.

Mayler G. A. Martins (S'10) received the B.S. degree in computer engineering in 2009 from the Federal University of Espírito Santo (UFES), Vitória, Brazil, and M.S. degree in microelectronics in 2012, from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, where he is currently pursuing the Doctor degree in microelectronics. In 2015, he is a visiting researcher at Universitat Politècnica de Catalunya (UPC), Catalonia, Spain.

His research interests include logic synthesis algorithms and emerging technologies.

Mr. Martins received a best paper award at SBCCI'13 conference.

André I. Reis (M'99–SM'05) received the B.S. degree in electrical engineering and the M.S. degree in computer science, both from Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1991 and 1993, respectively, and the Ph.D. degree in automatic and microelectronics systems from UMII, Montpellier, France, in 1998.

He is a Professor with the Institute of Informatics at UFRGS since 2000. He was a visiting researcher at University of Minnesota, Minneapolis, MN, USA, for the year 2004–2005. He has worked as a Chief Scientist for Nangate A/S, Herlev, Denmark, from July 2006 to February 2009. He has successfully coordinated UFRGS participation in the Synaptic FP7 European project consortium. He has 10 granted U.S. patents, and more than 150 academic publications.

Dr. Reis received best paper awards at the IFIP VLSI'97 and SBCCI'13 conferences. He is a member of Brazilian Microelectronics Society (SBMICRO), Brazilian Computer Society (SBC), and ACM. He is serving as the General Chair for IWLS 2015.

Renato P. Ribas (M'12) received the B.S. degree in electrical engineering from Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1991, the M.S. degree in electrical engineering from University of Campinas (Unicamp), Campinas, Brazil, in 1994, and the Doctor degree in microelectronics from the Institut National Polytechnique de Grenoble, France, in 1998.

Since 2000, he has been a Professor with the Institute of Informatics, Federal University of Rio Grande do Sul. He was a Post-Doctoral Researcher with the University of British Columbia, Vancouver, BC, Canada, in 2010, and Chief Scientist at Nangate A/S, Herlev, Denmark, in 2005. He has authored or co-authored more than 140 technical papers and he holds a U.S. patent. His research interest includes VLSI system and circuit design, computer-aided design development and new technologies for digital IC design.

Dr. Ribas received a best paper award at SBCCI'13 conference.