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 storing [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 re- sistor) [1]. The device resistance can be modified by applying a
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. 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. 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. 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 \( F(X) = B^n \rightarrow B \), where \( B = \{0, 1\} \). The constant functions zero and one are denoted by 0 and 1, respectively. In this work, we use \( \hat{f} \) to represent a Boolean expression for \( F \) over the basis \( \{\land, \lor, \neg\} \), where \( \{\land, \lor, \neg\} \) 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

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>next ( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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: \( V_{set}, V_{clear} \) or \( V_{cond} \). \( V_{set} \) (\( V_{clear} \)) is a voltage greater (smaller) than the memristor positive (negative) threshold, forcing the device into the low (high) resistance state. \( V_{cond} \) 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 \( V_p \) and \( V_q \), interact through a common node connected to a load resistor \( R_g \). Each memristor is a bipolar device with memristance \( M \). When \( M = M_{ON}(M = M_{OFF}) \), 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 \( V_q \) (\( V_p \)). Consider that \( V_{cond} \) is applied to \( V_p \) while \( V_{set} \) is applied to \( V_q \). If \( P \) is in high resistance state \( (p = 0) \), then \( V_p \) has negligible influence on the circuit, and \( V_{set} \) 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 \( V_p \) increases the voltage across \( R_g \), 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
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 \((V_p\) and \(V_q)\) and the load resistance \((R_g)\). 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 \(X_k(Y_k)\), 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 \(x_k \rightarrow y_j\), \(V\text{cond}\) is applied to \(X_k\) while \(V\text{set}\) 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 \(Y_h\) to the high resistance state by applying \(V\text{clear}\) to the target device. This operation is denoted by \(y_h = 0\). Usually, a reset operation is performed before a complementation operation on a variable 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

\[
f^* = x_1 \land \neg x_2.\tag{1}
\]

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, \(V\text{cond}\) is applied to memristor \(X_2\) and \(V\text{set}\) to

**Step 2:** This step writes \(x_2\) into work memristor \(Y_2\). Reset \(Y_2\). Apply \(V\text{cond}\) to \(Y_1\), and \(V\text{set}\) to \(Y_2\). The operations performed are: \(y_1 = 0\) and \(x_2 \rightarrow y_1\).

**Step 3:** This step performs a material implication from \(X_1\) to \(Y_2\). Apply \(V\text{cond}\) to \(X_1\) and \(V\text{set}\) 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 \(V\text{cond}\) to \(Y_2\) and \(V\text{set}\) 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**

<table>
<thead>
<tr>
<th>Step</th>
<th>(y_1)</th>
<th>(y_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>(\neg x_2)</td>
<td>-</td>
</tr>
<tr>
<td>Step 2</td>
<td>(\neg x_2)</td>
<td>(x_2)</td>
</tr>
<tr>
<td>Step 3</td>
<td>(\neg x_2)</td>
<td>(\neg x_2 \lor x_2)</td>
</tr>
<tr>
<td>Step 4</td>
<td>(x_1 \land \neg x_2)</td>
<td>(\neg x_1 \land x_2)</td>
</tr>
</tbody>
</table>

\(Y_1\), \(Y_1\) stores the complementary value of \(x_2(\neg x_2)\). The operations performed are: \(y_1 = 0\) and \(x_2 \rightarrow y_1\).

**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, \(V\text{cond}\) is applied to both \(X_1\) and \(X_2\), while \(V\text{set}\) is applied to both \(Y_1\) and \(Y_2\). Consequently, \(x_1 \rightarrow y_1\) and \(x_2 \rightarrow y_2\) 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 \(V\text{cond}\) to input memristors \(X_{k_0}, X_{k_1}, \ldots, X_{k_m}\), \(m \leq n\), while \(V\text{set}\) is applied to a work memristor \(Y_j\). A multi-input implication can be written as

\[
(x_{k_0} \lor x_{k_1} \lor \cdots \lor x_{k_m}) \rightarrow y_j, \quad j \in \{1, 2\}.\tag{3}
\]

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) \(\pi_k\), is a conjunction of variables \([40]\)

\[
\pi_k = \left\{1, S_k = \left\{\prod_{x_i \in S_k} x_i \mid S_k \in S(X) \setminus \emptyset\right\}\right.\tag{4}
\]

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

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$\pi_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\pi_8 = 1$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\pi_7 = x_3$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\pi_6 = x_2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\pi_4 = x_2 \land x_3$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\pi_5 = x_1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\pi_3 = x_1 \land x_3$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\pi_2 = x_1 \land x_2$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\pi_1 = x_1 \land x_2 \land x_3$</td>
</tr>
</tbody>
</table>

TABLE IV

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$\pi_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\pi_{16} = 1$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\pi_{15} = x_4$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\pi_{14} = x_3$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\pi_{11} = x_3 \land x_4$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\pi_{13} = x_2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\pi_{10} = x_2 \land x_3$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\pi_9 = x_2 \land x_3$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\pi_5 = x_2 \land x_3 \land x_4$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\pi_{12} = x_1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\pi_8 = x_1 \land x_4$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\pi_7 = x_1 \land x_3$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\pi_4 = x_1 \land x_3 \land x_4$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\pi_6 = x_1 \land x_2$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\pi_3 = x_1 \land x_2 \land x_4$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\pi_2 = x_1 \land x_2 \land x_3$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\pi_1 = x_1 \land x_2 \land x_3 \land x_4$</td>
</tr>
</tbody>
</table>

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

$$\neg x_{k_1} \lor \neg x_{k_2} \lor \cdots \lor \neg x_{k_m} \rightarrow y_j, \quad j \in \{1, 2\}.$$  (5)

Since $x_{k_1} \land x_{k_2} \land \cdots \land x_{k_m}$ equals a $\pi_k$, (5) can be written as

$$\neg \pi_k \rightarrow y_j.$$  (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

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

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

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

where $\otimes$ denotes the exclusive disjunction operation and each $\beta_k$ is a binary $n$-tuple $\beta_k = \{b_{k_1}, b_{k_2}, \ldots, b_{kn}\}$ satisfying

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

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

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

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

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

$$f^* = x_1 \land x_2 \lor \neg x_3.$$  (12)

A recursive IMPLY expression for $F$ is

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

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

$$\neg \pi_k = \neg \pi_3$$  (14a)

$$f_1 = (\neg \pi_7 \rightarrow 0) \rightarrow 0.$$  (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 \([\pi_j, \pi_k]\) if \(f(\pi_j) \subset \pi_k\). Each vertex is colored according to the Boolean function value for the minterm related to this vertex (e.g., minterm \(x_1 \land \neg x_2 \land \neg x_3\) is associated to PPT \(\pi_5 = x_1\)). The graph is partitioned in such a way that all vertices 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:

\[
(15)
\]

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

\[
(17)
\]

where the PPT for \(\pi\) are \(\pi_0\), \(\pi_1\), \(\pi_2\), and \(\pi_3\). The direct correction yields an expression with eight operators, as follows:

\[
(18)
\]

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

\[
(19)
\]

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 \(V_{\text{cond}}\) to a work memristor and to a set of input memristors, while \(V_{\text{set}}\) is applied to the other work memristor. This generalization enables operations in the following form to be computed in a single step:

\[
(20)
\]

Alternatively, (21) can be rewritten as follows:

\[
(21)
\]

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

\[
(22)
\]

where \(f_1\) and \(f_2\) are IMPLY expressions. If \(f_1\) 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 ‘\(\lor\)’. 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

\[
(23)
\]

where \(f_1\) is in the form of (23). Notice that (23c) becomes (11c) when \(\neg \pi_{\text{cond}} = 0\). Also, (23c) corresponds to (22) with \(f_2 = 0\). Allowing \(f_2 \neq 0\) 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:

\[
(24)
\]

where \(\pi_0\) and \(\pi_1\) are conjunctions of literals given by

\[
(25)
\]

and

\[
(26)
\]

where \(\rho\) and \(\eta\) are the number of elements of \(X_p\) and \(X_n\), respectively. Since any conjunction of positive literals corresponds to a PPT, \(f_p^*\) equals a PPT \(\pi_{fp}\). In turn, \(f_n^*\) can be written as function of PPT as

\[
(27)
\]

An IMPLY expression is obtained as follows:

\[
(28)
\]
which requires \( \eta + 3 \) operations. Each PPT in \( f n^* \) adds an IMPLY operation. The term \( \pi_{p}\) contributes with two implications and the final complementation operation adds another implication. The number of operations does not depend on \( \rho \) because any PPT is computed in a single step. If multi-memristor implication is used, \( f \) becomes

\[
f = (\neg f_p \lor (\neg \pi_{n_1} \rightarrow \cdots \rightarrow (\neg \pi_{n_1} \rightarrow 0) \cdots)) \rightarrow 0 \tag{29}
\]

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

\[
f_1 = \neg \pi_{n_1} \rightarrow \cdots \rightarrow (\neg \pi_{n_2} \rightarrow (\neg \pi_{n_1} \rightarrow 0)) \tag{30}
\]

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

\[
f^* = x_1 \land x_2 \land \neg x_3 \land \neg x_4. \tag{31}
\]

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

\[
f = (\neg \pi_{15} \rightarrow (\neg \pi_{14} \rightarrow ((\neg \pi_6 \rightarrow 0) \rightarrow 0))) \rightarrow 0. \tag{32}
\]

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

\[
f = (\neg \pi_6 \lor (\neg \pi_{15} \rightarrow (\neg \pi_{14} \rightarrow 0))) \rightarrow 0 \tag{33}
\]

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:** \( y_2 = 0 \), \( (\neg \pi_6 \lor y_1) \rightarrow y_2 \). To perform the last operation, apply \( Vcond \) to \( Y_1 \) and to all input memristors in \( \neg \pi_6 \) (\( X_2 \) 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

\[
f = \begin{cases} 
  0 & \text{if } f_1 \text{ and } f_2 \text{ are given by (34)} \\
  \neg \pi_6 \rightarrow f_1 & \text{Notice that (34b) and (34c) are reduced to (23b) and (23c), respectively, when } f_2 = 0. \\
  (\neg \pi_6 \lor f_1) \rightarrow f_2 & \text{Hence, any recursive form is a factored form though not all factored forms are a recursive one. Since factored forms are more general than recursive ones, it may be possible to obtain smaller IMPLY expressions. However, not all factored forms can be computed with two work memristors. Therefore, the number of work memristors necessary to compute a Boolean function becomes a concern.}
  f_1 \rightarrow f_2 & \text{In the following, we discuss which factored forms can be computed with two work memristors. Moreover, a simple method to determine the number of work memristors necessary to compute an IMPLY expression is presented. We use } \mu(f) \text{ to denote the number of work memristors required to compute an IMPLY expression } f.
\end{cases} \tag{34}
\]

The number of work memristors to compute (34b) equals to the maximum between one and \( \mu(f_1) \). Expressions given by (34c) or (34d) can be computed using two work memristors if \( \mu(f_1), \mu(f_2) \) is smaller or equal to two and \( \mu(f_2), \mu(f_1) \) is at most one (i.e., \( \mu(f_1) + \mu(f_2) \leq 3 \)). In this case, the function requiring more work memristors must be computed first. The reasoning for this argument is as follows. Let \( \mu(f_1) = 1 \) and \( \mu(f_2) = 2 \). Compute \( f_2 \) using both work memristors available. Once \( f_2 \) is evaluated, one work memristor must store the resulting value. However, the other work memristor, can be used to calculate \( f_1 \). After \( f_1 \) is computed, each work memristor stores either \( f_1 \) or \( f_2 \) and the last IMPLY can be performed. On the other hand, if \( f_1 \) is evaluated before \( f_2 \), a third work memristor must be added because there is only one work memristor available to compute \( f_2 \). If \( \mu(f_1) = \mu(f_2) = 2 \), three work memristors are required regardless of the order chosen to compute \( f_1 \) and \( f_2 \). 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 \( f_1 \) and \( f_2 \) as follows:

\[
\mu(f) = \begin{cases} 
  \max(\mu(f_1), \mu(f_2)) & \text{if } \mu(f_1), \mu(f_2) \text{ are different, the greater value between them is chosen. In this case, the function that requires more work memristors must be calculated before the other one. When } \mu(f_1) \text{ and } \mu(f_2) \text{ are equal, a work memristor must be added, regardless the ordering chosen to compute } f_1 \text{ and } f_2.
  \mu(f_1) = \mu(f_2) + 1 & \text{regardless the ordering chosen to compute } f_1 \text{ and } f_2.
\end{cases} \tag{35}
\]

If \( \mu(f_1) \) and \( \mu(f_2) \) 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 \( \mu(f_1) \) and \( \mu(f_2) \) are equal, a work memristor must be added, regardless the ordering chosen to compute \( f_1 \) and \( f_2 \).

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

\[
f = (\neg \pi_6 \rightarrow ((\neg \pi_a \rightarrow 0) \rightarrow 0)) \rightarrow (\neg \pi_d \rightarrow ((\neg \pi_c \rightarrow 0)) \tag{36}
\]

where \( \pi_a, \pi_b, \pi_c, \text{ and } \pi_d \) are arbitrary PPT. Expression (36) is in the form of (34c), being

\[
f_1 = (\neg \pi_6 \rightarrow ((\neg \pi_a \rightarrow 0) \rightarrow 0)); \tag{37a}
\]

\[
f_2 = (\neg \pi_d \rightarrow ((\neg \pi_c \rightarrow 0) \rightarrow 0)). \tag{37b}
\]

The last IMPLY operation \( f_1 \rightarrow f_2 \) 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
include a third work memristor in Step 5 (similar analysis holds when \( f_1 \) is computed before \( f_2 \)).

Step 1: \( y_1 = 0, -\pi_a \rightarrow y_4, Y_1 \) stores \( \pi_a \).

Step 2: \( y_2 = 0, y_1 \rightarrow y_2, Y_2 \) stores \( -\pi_a \).

Step 3: \( -\pi_b \rightarrow y_2, Y_2 \) stores \( \pi_b \lor -\pi_a \). This step concludes the computation of \( f_1 \).

Step 4: \( y_1 = 0, -\pi_c \rightarrow y_1, Y_3 \) stores \( \pi_c \). 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 \( f_1 \) and cannot be used to keep the resulting value. The solution is to add a work memristor \( Y_3 \).

Step 6: \( -\pi_d \rightarrow y_2, Y_2 \) stores \( \pi_d \lor -\pi_a \). This step concludes the computation of \( f_2 \), which is stored on \( Y_2 \).

Step 7: \( y_3 \rightarrow y_3 \). This step performs the last IMPLY \( (f_1 \rightarrow f_2) \). 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 \( f_1 \) and \( f_2 \) require two work memristors. Expression \( f_1 \) uses \( Y_1 \) and \( Y_2 \) whereas \( f_2 \) uses \( Y_3 \) and \( Y_3 \). As expected from (35), three work memristors are required to compute \( f \).

Table V

<table>
<thead>
<tr>
<th>( Y_1 )</th>
<th>( Y_2 )</th>
<th>( Y_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: ( \pi_a )</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Step 2: ( \pi_a )</td>
<td>( -\pi_a )</td>
<td>-</td>
</tr>
<tr>
<td>Step 3: ( \pi_a )</td>
<td>( \pi_b \lor -\pi_a )</td>
<td>-</td>
</tr>
<tr>
<td>Step 4: ( \pi_c )</td>
<td>( \pi_b \lor -\pi_a )</td>
<td>( -\pi_c )</td>
</tr>
<tr>
<td>Step 5: ( \pi_d \lor \pi_c )</td>
<td>( \pi_b \lor -\pi_a )</td>
<td>( -\pi_c )</td>
</tr>
<tr>
<td>Step 6: ( -\pi_b \land \pi_d \lor \pi_c )</td>
<td>( \pi_b \lor -\pi_a )</td>
<td>( -\pi_c )</td>
</tr>
</tbody>
</table>

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

Step 5: \(-\pi_d \rightarrow y_1 \). This step concludes the computation of \( f_3 \), 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 \( (f_1 \rightarrow f_3) \). \( Y_1 \) keeps the final result.

Since \( \mu(f_1) = 2 \) and \( \mu(f_3) = 1 \), two work memristors suffice to compute (38). Table VI presents the state of work memristors at the end of each step.

Table VI

<table>
<thead>
<tr>
<th>( Y_1 )</th>
<th>( Y_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: ( \pi_a )</td>
<td>-</td>
</tr>
<tr>
<td>Step 2: ( \pi_a )</td>
<td>( -\pi_a )</td>
</tr>
<tr>
<td>Step 3: ( \pi_a )</td>
<td>( \pi_b \lor -\pi_a )</td>
</tr>
<tr>
<td>Step 4: ( \pi_c )</td>
<td>( \pi_b \lor -\pi_a )</td>
</tr>
<tr>
<td>Step 5: ( \pi_d \lor \pi_c )</td>
<td>( \pi_b \lor -\pi_a )</td>
</tr>
<tr>
<td>Step 6: ( -\pi_b \land \pi_d \lor \pi_c )</td>
<td>( \pi_b \lor -\pi_a )</td>
</tr>
</tbody>
</table>

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

\[
\{ -\pi_2 \lor ( -\pi_3 \rightarrow 0) \} \rightarrow \{ ( -\pi_3 \lor ( -\pi_2 \rightarrow 0) \} \rightarrow 0 \}. \tag{40}
\]

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

\[
f_1 = -\pi_3 \rightarrow 0 \quad \tag{41a}
\]

\[
f_2 = (-\pi_3 \lor (-\pi_2 \rightarrow 0)) \rightarrow 0. \tag{41b}
\]

Since \( \mu(f_1) = 2 \) and \( \mu(f_2) = 2 \), (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 \( ( -\pi_k ) \), including the constant value 0. This choice is made because Boolean expressions are written in terms of \( -\pi_k \). Consequently, PPT comprise 1-cost functions because one IMPLY is needed to implement these functions \( ( -\pi_k \rightarrow 0 ) \). 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
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

\[
\begin{align*}
\text{(42a)} & \\
\text{(42b)} & 
\end{align*}
\]

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 calculated. 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):

\[
\begin{align*}
\text{(43a)} & \\
\text{(43b)} & 
\end{align*}
\]

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.

2) Example 8: Consider a function \( f \) described by

\[
\begin{align*}
\text{f}^* = \neg \pi_2 \lor \pi_2. 
\end{align*}
\]

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 \( C_5 \), as follows:

\[
\begin{align*}
f_3 &= \neg \pi_3 \rightarrow \{\neg \pi_2 \rightarrow 0\} \rightarrow 0. 
\end{align*}
\]

Hence the algorithm stops without creating sets \( C_4 \) and \( C_5 \).

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

\[
\begin{align*}
\text{(46a)} & \\
\text{(46b)} & 
\end{align*}
\]

Notice that (46b) is reduced to (42b), when \( \pi_{c-1} \). 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

\[
\begin{align*}
\text{(47a)} & \\
\text{(47b)} & 
\end{align*}
\]

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, \( f_{1a} \) and \( f_{1b} \) [similar expressions are obtained for (47b)]

\[
\begin{align*}
f_{1a} &= \{\neg \pi_3 \rightarrow \{\neg \pi_3 \rightarrow 0\} \rightarrow 0\} \rightarrow 0 \\
f_{1b} &= \{\neg \pi_3 \lor \{\neg \pi_2 \rightarrow 0\} \rightarrow 0\} \rightarrow 0 
\end{align*}
\]
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:

\[
\begin{align*}
\{f_c & \rightarrow f_{c-1} \} \quad (49a) \\
\{f_{c-1} & \rightarrow 0 \} \quad (49b) \\
\{f_{c-a} \rightarrow f_{a-1}, \mu(f_{c-a}) + \mu(f_{a-1}) \leq 3 \} \quad (49c)
\end{align*}
\]

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

\[
\bar{f}^* = \{\neg x_1 \land \neg x_2 \} \lor \{\neg x_3 \land \neg x_4\}. \quad (50)
\]

Following the same procedure described in previous examples, a multi-input implication factored form \(f(1)\) for \(F\) appears in \(C_2\), 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

\[
\begin{align*}
f_{c-a} = f_2 &= \{\neg \pi_{14} \rightarrow (\neg \pi_{15} \rightarrow 0)\} \quad (52a) \\
f_{a-1} = f_4 &= \{\neg \pi_{12} \rightarrow ((\neg \pi_{15} \rightarrow 0) \rightarrow 0)\} \rightarrow 0]. \quad (52b)
\end{align*}
\]

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

\[
\begin{align*}
f_0 &= \neg \pi_{14} \quad (53a) \\
f_{c-1} &= f_1 = \neg \pi_{15} \rightarrow 0. \quad (53b)
\end{align*}
\]

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 (\(f_2\)) for \(F\) has eight operations

\[
\begin{align*}
f_2 &= \{\neg \pi_8 \rightarrow (\neg \pi_7 \rightarrow ((\neg \pi_{13} \rightarrow f_2a) \rightarrow 0))\} \rightarrow 0 \quad (55a) \\
f_2a &= \{(\neg \pi_{15} \rightarrow (\neg \pi_{14} \rightarrow 0)\} \rightarrow 0 \quad (55b)
\end{align*}
\]

where \(\neg \pi_7 = \neg (x_1 \land x_2)\) and \(\neg \pi_8 = \neg (x_1 \land x_4)\). Hence the utilization of a factored form, computable with 2 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:

\[
f_c = \begin{cases} 
\{f_0 \rightarrow f_{c-1} \} \\
\{f_{c-1} \lor f_0 \} \rightarrow 0 \\
\{f_{c-a} \lor f_{a-1}, \mu(f_{c-a}) + \mu(f_{a-1}) \leq 3 \}
\end{cases} \quad (56a, 56b, 56c)
\]

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 two-input Boolean function:

\[
\begin{align*}
f^* &= \neg x_1 \land (x_2 \land x_3). \quad (57)
\end{align*}
\]

By setting \(c = 2, 3, 4\) 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

\[
\begin{align*}
f_2 &= \{f_0 \rightarrow f_1 \} \rightarrow (f_1 \lor f_0) \rightarrow 0 \quad (58a) \\
f_3 &= \{f_0 \rightarrow f_2 \} \rightarrow (f_2 \lor f_0) \rightarrow 0 \quad (59a) \\
f_4 &= \{f_0 \rightarrow f_3 \} \rightarrow (f_3 \lor f_0) \rightarrow 0 \quad (60a) \\
f_4 &= \{f_1 \lor f_0 \rightarrow f_1 \} \rightarrow 0 \quad (60c)
\end{align*}
\]

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 \(c = 3, 4\) can only assume the value 2. Finally, for \(c = 4, a\) can either be 2 or 3.

After generating sets \(C_2, C_3\) the best implementation for \(F(f)\) is found with 4 material implications, as follows:

\[
\begin{align*}
f_1 &= (\{(\neg \pi_{15} \rightarrow 0) \lor \neg \pi_6\} \rightarrow (\{(\neg \pi_7 \lor \neg \pi_{15} \rightarrow 0)\} \rightarrow 0). \quad (61)
\end{align*}
\]

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

\[
\begin{align*}
f_0 &= \neg \pi_6 \quad (62a) \\
f_{c-a} &= f_1 = \neg \pi_6 \rightarrow 0 \quad (62b) \\
f_{a-1} &= f_2 = (\neg \pi_7 \lor \neg \pi_{15} \rightarrow 0) \rightarrow 0. \quad (62c)
\end{align*}
\]

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

\[
\begin{align*}
f_{c-1} &= f_1 = \neg \pi_5 \rightarrow 0 \quad (63a) \\
f_0 &= \neg \pi_7. \quad (63b)
\end{align*}
\]

The best recursive IMPLY expression for \(F(f)_2\) has five operators

\[
\begin{align*}
f_2 &= (\neg \pi_5 \rightarrow (\{(\neg \pi_6 \rightarrow (\neg \pi_7 \rightarrow 0)\} \rightarrow 0)) \rightarrow 0. \quad (64)
\end{align*}
\]

VIII. EXPERIMENTAL RESULTS

As presented in [39] and [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

<table>
<thead>
<tr>
<th></th>
<th>Average number of implications</th>
<th>Normalized average</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>8.83</td>
<td>1.00</td>
</tr>
<tr>
<td>[40]</td>
<td>8.91</td>
<td>1.01</td>
</tr>
<tr>
<td>[39]</td>
<td>12.00</td>
<td>1.35</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Multi-input implication</th>
<th>Multi-memristor implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive</td>
<td>9.26</td>
<td>8.85</td>
</tr>
<tr>
<td>Factored</td>
<td>9.24</td>
<td>8.14</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>Reduction</th>
<th>Multi-memristor implication (recursive)</th>
<th>Multi-memristor implication (factored)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49,078</td>
<td>21,703</td>
</tr>
<tr>
<td>1</td>
<td>6,240</td>
<td>19,745</td>
</tr>
<tr>
<td>2</td>
<td>10,120</td>
<td>19,354</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3,981</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
<td>664</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>

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


Felipe S. Marranghello

Vinicius Callegaro

SBCCI’13 conference

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on November 28,2023 at 00:00:38 UTC from IEEE Xplore. Restrictions apply.
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 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 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.