An Improved Algorithm for IMPLY Logic Based Memristive Full-Adder

Shokat Ganjeheizadeh Rohani and Nima TaheriNejad
Institute of Computer Technology
TU Wien, Vienna, Austria
Email: shokouh.ganjei@gmail.com, nima.taherinejad@tuwien.ac.at

***Abstract*—Memristor characteristics like high speed, low power, and passive memory retention make it suitable for application in several fields. Neuromorphic systems, digital and analog circuits, or simply a memory unit, are some of these applications. A memristor exhibits different properties, which allow each field of application to take the desired advantages of this device. Memristor-based "Implication", which implements the IMPLY logic using material properties of memristors, makes all logic operations possible in designs, which consists of only memristors as fundamental constituents. The focus of this article is on memristor-based Full-adder, which uses only the IMPLY logic. We propose an algorithm with the merit of needing fewer execution steps and a smaller number of memristors.**

I. INTRODUCTION

Memristor, known as the fourth circuit element, is a two terminal device, which establishes a relation between electric flux (charge) and the magnetic flux [1]. This relation is exhibited by changing the resistance of this component, as the current passes through it. The constitutive equations of memristor result in memristance and memductance [2], [3]. They imply that this circuit element has a memory effect based on the history of the past current or voltage applied to it.

The resistance of the memristor (a thin layer of metal-oxide, e.g., TiO_2, with two differently doped regions sandwiched between two conductors [4]) alters between its minimum (R_{on}) and its maximum (R_{off})[1] as the bias voltage changes. By increasing the voltage in positive bias[2], the dopant (charge carriers) diffuse from the doped zone of memristor to the undoped side and as a consequence the length of the doped area extends, which leads to a smaller resistance. The case $R = 0$, or R_{on}, is valid when the whole length is filled with the doped area and the case $R = 1$, or R_{off}, happens in negative biasing, when the whole memristor length becomes undoped [5]. If memristor is used in digital circuits, the logical states 0 and 1 are represented respectively by Off- and On-states and considering a tolerance range in between.

Among various applications [6] for memristors, logic circuits implemented with "only memristors" is the topic of our work. This approach provides a structure which can replace the classical "Von Neumann architecture" [7]–[10]. We focus on Full-adders, due to the fact that they are a very basic component of many digital circuits and improving their performance can improve many processing systems.

[1]Or in other words, within resistance interval of $[0, 1]$, if we normalize the memristance to its maximum.

[2]Positive bias is when the doped area has a higher voltage than the undoped area and for negative biasing it is the other way around.

TABLE I: Truth Table of IMPLY Logic & its Memristor States.

a	b	R_a	R_b	$b' = a \to b$	$R_{b'}$
0	0	R_{off}	R_{off}	1	R_{on}
0	1	R_{off}	R_{on}	1	R_{on}
1	0	R_{on}	R_{off}	0	R_{off}
1	1	R_{on}	R_{on}	1	R_{on}

The remainder of this paper is organized as follows; An introduction to IMPLY and converting other Boolean logics to IMPLY is given in Section II. In the same section, we review logic minimization and existing full-adder implementations using IMPLY logic. In Section III, lay our contributions; our proposed algorithm and the equivalencies in IMPLY logic, which helped us in developing the new algorithm. To put our work in perspective, with respect to other similar works, we have included comparison and some discussions in Section III. Finally, Section IV concludes the paper.

II. BACKGROUND

A. IMPLY Logic Structure

The natural structure, physics and other characteristics of memristor make it very compatible with performing the IMPLY, rather than Boolean logic [11]. The truth table of IMPLY logic is shown in Table I, where $a \to b$ represents a implies b. The end value will be saved in b, which means that this memristor loses its initial value. We can see that the IMPLY has for all combinations of the two inputs, a and b, an output value of 1, except for $a = 1$ and $b = 0$. IMPLY logic can be implemented as a circuit with two memristors, which are connected to a conventional resistor. Memristors have the role of digital switches and input values are the initial resistance (state) of them. Fixed voltages are applied to the memristors, and after applying these fixed voltages, the output is the final state of the memristor connected to the voltage with the higher magnitude. The design procedure of IMPLY logic using memristors is described in details in other works, for instance, see [12].

Although, IMPLY logic is not the only solution for implementing memristive Full-adders (for an example see [13]), it is an attractive option because it can lead to memristor-only structures, which are very compatible with the crossbar array design. This design is widely used in neuromorphic networks, FPAAs and memories [6]. In such a structure, other logic operations, can be performed using IMPLY and FALSE [14]. In which FALSE, is a function that always yields zero at the output. One possible solution is shown in Table II.

978-1-5090-5538-8/17/$31.00 ©2017 IEEE

TABLE II: Implementing Boolean Logics using IMPLY

Logic Operations	Implementation Based on IMPLY Logic
NOT a	$a \to 0 \quad \equiv \quad a \to \text{FALSE}$
a OR b	$(b \to 0) \to a$
a NOR b	$\big((b \to 0) \to a\big) \to 0$
a NAND b	$b \to (a \to 0)$
a AND b	$\big(b \to (a \to 0)\big) \to 0$
a XOR b	$\big\{\big((a \to 0) \to b\big) \to \big[\big(a \to (b \to 0)\big) \to 0\big]\big\} \to 0$

B. Existing Algorithms and Logic Minimization

Full-adder can be demonstrated as a function, $f(3,2)$, that is, a function with 3 inputs (two operands and one Carry-in) and two outputs (one Sum and one Carry-out). For an n-bit Full-adder with a, b and c as input operands, a set of a and b memristors ($a = \{a_1, a_2, ..., a_n\}$ and $b = \{b_1, b_2, ..., b_n\}$) and one memristor for Carry-in as input, as well as a set of work memristors ($s = \{s_1, ..., s_n\}$) are needed (Although, theoretically, for any number of inputs, two work memristors are sufficient for computing any Boolean function [15]). Work memristors are used for copying the inputs to them, saving the outputs, or conducting some computations on them. For the output, $n+1$ memristors are needed (one for Carry-out and n for Sum). Some of the existing logic minimization and methods for implementing a Full-adder using memristors, as well as their performance report for a single bit adder, are as it follows:

1) *Conjunctive Normal Form (CNF):*

$$\begin{aligned} S =& (c_{in} + a + b).(\overline{c}_{in} + a + \overline{b}).(c_{in} + \overline{a} + \overline{b}). \\ & (\overline{c}_{in} + \overline{a} + b) \\ C_{out} =& (c_{in} + a).(c_{in} + b).(a + b) \end{aligned}$$

For building one bit Full-adder with the help of CNF, as it is explained in [11], 89 total steps (48 steps for calculating the Sum, 39 steps for the Carry-out and two additional steps for initializing two of the work memristors) are needed and four work memristors are used[3].

2) *Disjunctive Normal Form (DNF), implemented using XOR logic:*

$$\begin{aligned} S &= (c_{in}.a.b) + (\overline{c}_{in}.a.\overline{b}) + (c_{in}.\overline{a}.\overline{b}) + (\overline{c}_{in}.\overline{a}.b) \\ &= c_{in} \oplus (a \oplus b) \\ C_{out} &= (\overline{c}_{in}.a.b).(c_{in} + a + \overline{b}).(c_{in} + \overline{a} + b) \\ &\quad + (c_{in}.a.b) = a.b + c_{in}.(a \oplus b) \end{aligned}$$

The improved Full-adder which is built with XOR logic [16], needs 29 steps and three work memristors. In this work, it is shown that 13 steps are necessary for each XOR. Since two XORs are needed for calculating the Sum, then as a result, 26 steps will be needed for calculating the Sum, and 3 extra steps for the Carry.

In [14], the authors have considered only the number of IMPLYs as delay for the output and tried to reduce the delay time. In the end, it needed 19 cycles for calculating the Sum and 18 cycles for the Carry. This article has an assumption, that the clearing of work memristor s, occurs just at the beginning and the device will not be cleared in subsequent steps. In other words, in order to avoid clearing the s in other steps, we will need more memristors. The improved Full-adder circuit mentioned in [17], needs 20 steps for computing the Sum and three more steps for the Carry (23 in total), and 3 extra memristors in addition to input memristors (a, b and c_{in}).

[3]We note that, as mentioned before, theoretically two work memristors could be sufficient, however, this would increase the number of steps and consequently slow down the adder.

TABLE III: Truth Table of Full-Adder

c_{in}	a	b	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

C. Serial vs. Parallel Implementation

There are two approaches for the implementation of a Full-adder; serial and parallel. Most approaches [11], [16], [17] use serial based implementation, in which all memristors are in the same row. This is the standard structure for the crossbar design. In serial implementation, since all memristors including input, output and work memristors are in one row, only one operation (IMPLY or FALSE) can be performed in each clock cycle. More recently, a parallel approach was introduced by Kvatinsky et al. [16]. In parallel model each bit stands in a different row with its related work memristors. The advantage of this model is that all independent operations (such as FALSE operation or operations which don't use the results from other steps) can be executed simultaneously. The parallel design, in which each bit of the Full-adder is in a different row, has some disadvantages such as; (i) Increasing the number of memristors to 9 for each bit ($9N$), (ii) some operations can not be executed until the former bit is finished with the Carry-out result, (iii) Due to the fact that each row works in serial and each operation related to a row should be performed in one cycle, only independent operations in different rows can be performed simultaneously, (iv) its structure differs from that of the standard crossbar design.

Therefore, given that serial structure is easier to implement (compatible with crossbar), and is more standard, in this work, we focus on the serial design. Lastly, we note that for improving the functionality of a Full-adder a compromise between two factors, namely number of memristors (area) and computation steps (speed), should be considered.

III. Proposed Full-adder Algorithm

A. Logic Minimization

In order to achieve a more effective implementation for IMPLY based Full-adder, the IMPLY statements should be minimized as much as possible and the appropriate equivalent statements for implementation should be chosen. Here are ten of these equivalencies in IMPLY logic, which we worked out and used to minimize the necessary steps for implementing the Full-adder algorithm.

1. $a \to (b \to c) \equiv b \to (a \to c)$
2. $(a \to \overline{b}) \to \overline{a} \equiv (\overline{a} \to b) \to b \equiv a \to b$

3. $(a \to \bar{b}) \to \bar{b} \equiv (\bar{a} \to b) \to a \equiv b \to a$
4. $a \to \bar{a} \equiv \bar{a}$ *and* $a \to a \equiv a$
5. $b \to a \equiv \bar{a} \to \bar{b}$ *and* $a \to b \equiv \bar{b} \to \bar{a}$
6. $\bar{a} \to b \equiv \bar{b} \to a$ (Equivalent to OR)
7. $a \to \bar{b} \equiv b \to \bar{a}$ (Equivalent to NAND)
8. $(a \to b) \to \overline{(b \to a)} \equiv (a \to b) \to \overline{(\bar{a} \to \bar{b})} \equiv$ $\overline{(\bar{a} \to b) \to \overline{(a \to \bar{b})}}$ (Equivalent to XOR)
9. $\overline{(\bar{a} \to b)} \equiv \overline{(\bar{b} \to a)}$ (Equivalent to NOR)
10. $\overline{(a \to \bar{b})} \equiv \overline{(b \to \bar{a})}$ (Equivalent to AND)

Working backwards, from the output to the inputs, we need to find two statements, whose implication results directly in Carry-out and Sum, as shown in the truth table of the adder, Table III. That is, to find two statements P and Q, so that the result of their implication would be Carry-out or Sum; e.g., $C_{out} = P \to Q$. The following shows our findings where the bit-stream pattern inside brackets corresponds the top-down bit-stream pattern in Table III.

*1) **Carry-out** = $[00010111]$:* According to the truth table of IMPLY (Table I), if the first operand (antecedent) is zero or the second operand (consequent) is 1 then the output is certainly 1. However, in order to have a 0 output, the antecedent must be 1, which determines the pattern for P. Therefore, to find the desired statements, P should have a structure like, $P = [111X1XXX]$, where X shows "don't care". Seven potential solutions are shown in Equation 1. It should be mentioned that each statement, which stands for each outcome, is just one of the possibilities of equivalent terms.

$$\begin{aligned}
&P = [111X1XXX] \\
&P_1 = [11101010] \quad : \quad (\bar{a} \to c) \to \bar{b} \\
&P_2 = [11101100] \quad : \quad (\bar{b} \to c) \to \bar{a} \\
&P_3 = [11111000] \quad : \quad (\bar{a} \to b) \to \bar{c} \\
&P_4 = [11111101] \quad : \quad c \to (a \to b) \\
&P_5 = [11101111] \quad : \quad b \to (a \to c) \\
&P_6 = [11111011] \quad : \quad c \to (b \to a) \\
&P_7 = [11111110] \quad : \quad c \to (b \to \bar{a})
\end{aligned} \tag{1}$$

The corresponding terms for Q are as it follows (numbers refer to the statements in Table IV):

$$\begin{aligned}
&Q_1 = [000X0X1X] \quad : \quad (2, 5, 6) \\
&Q_2 = [000X01XX] \quad : \quad (4, 6, 3) \\
&Q_3 = [00010XXX] \quad : \quad (1, 4, 5) \\
&Q_4 = [000101X1] \quad : \quad (4) \\
&Q_5 = [000X0111] \quad : \quad (6) \\
&Q_6 = [00010X11] \quad : \quad (5) \\
&Q_7 = [0001011X] \quad : \quad -
\end{aligned} \tag{2}$$

*2) **SUM** = $[01101001]$:* Carrying out the same procedure for the Sum, we found the following solutions:

TABLE IV: Q Statements

1	$\overline{a \to (\bar{c} \to \bar{b})}$	2	$\overline{\bar{b} \to (a \to \bar{c})}$	3	$\overline{c \to (b \to a)}$
4	$\overline{(\bar{a} \to c) \to \bar{b}}$	5	$\overline{(\bar{b} \to c) \to \bar{a}}$	6	$\overline{(\bar{a} \to b) \to \bar{c}}$

$$\begin{aligned}
&P = [1XX1X11X] \\
&P_1 = [11011111] \quad : \quad \bar{b} \to (a \to c) \\
&P_2 = [10111111] \quad : \quad b \to (\bar{a} \to c) \\
&P_3 = [11110111] \quad : \quad c \to (\bar{a} \to b) \\
&P_4 = [11111110] \quad : \quad c \to (b \to \bar{a}),
\end{aligned} \tag{3}$$

in which corresponding Q terms are:

$$\begin{aligned}
&Q_1 = [01X01001] \\
&Q_2 = [0X101001] \\
&Q_3 = [0110X001] \\
&Q_4 = [0110100X]
\end{aligned} \tag{4}$$

However, we were not able to find any statements among combinations with three inputs, which could match these outputs directly. Therefore, the next step is to find a more complex and at the same time minimized term for the needed values. Using Karnaugh map we found some statements, from which those containing the same terms as Carry-out were selected in order to minimize the steps for implementing the Full-adder.

B. Improved Full-adder Algorithm

Here, we propose a method, which benefits from the advantages of both smaller foot-print and smaller number of steps. Proposed approach for computing Sum and Carry-out is presented in logical statements of Equation (5) and Equation (6), respectively. The precise algorithm that uses these statements for calculating the Sum and Carry-out is spelled out in Table VI. The proposed algorithm can output the results in 22 computational steps, using only 2 work memristors (i.e., the minimum number of work memristors possible).

$$S = \Big[(\bar{a} \to b) \to \Big((a \to \bar{b}) \to c\Big)\Big] \to \Big(\overline{\overline{(a \oplus b)} \to \bar{c}}\Big) \tag{5}$$

$$C_{out} = \overline{\Big[(\bar{a} \to b) \to \overline{\Big((a \to \bar{b}) \to c\Big)}\Big]} \tag{6}$$

Advantages of the proposed algorithm are:

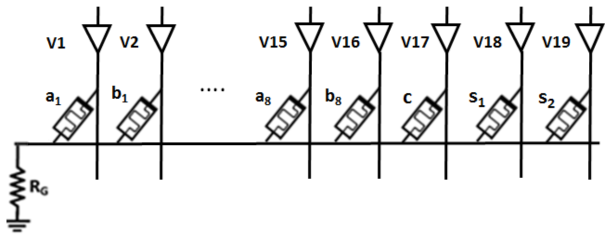

Fig. 1: Serial structure for 8-bit full adder

TABLE V: Summary of comparisons between the proposed algorithm and other similar works for $n = 8$ (an 8-bit Full-adder).

Algorithm	Number of Memristors						Number of Steps			
	Input	Output	Reused	Work	Total	Improv.	IMPLY	FALSE	Total	Improv.
[16]	$2n + 1$	$n + 1$	1	2	$3n + 3$ (27)	30%	$19n$	$10n$	$29n$ (232)	25%
[17]	$2n + 1$	$n + 1$	1	2	$3n + 3$ (27)	30%	$15n$	$8n$	$23n$ (184)	5%
Proposed	$2n + 1$	$n + 1$	$n + 1$	2	$2n + 3$ (19)	-	$15n$	$7n$	$22n$ (176)	-

1) Speed: In serial application the computation needs 22 Steps, instead of 23 steps in the best previous work [17].

2) Area: The Carry-out result will be saved on the same memristor as Carry-in (c memristor) and in addition to that, the Sum will be saved on the same memristor as the input (a memristor), which reduces the need for an extra output memristor. Total number of memristors for an n bit Full-adder is $2n+3$ (e.g., 19 for 8 bit), which is n memristor less than the previous works [16], [17]. Number of needed work memristors for one bit is 2, in comparison to 3 in [17] and [16].

As mentioned, designing a Full-adder is a compromise between area and speed (calculation time). Hence, the merit of the algorithms are determined based on those two criteria, namely the number of needed memristors and the number of execution steps. In Table V, these measures are evaluated and compared for this work and two other related recent works in the literature. In this table, the percentage of improvement (for an 8-bit implementation; $n = 8$) in comparison with other works is also given.

We observe that the proposed algorithm in this work shows a positive improvement in both aspects of merit. That is, 30% better in number of memristors (which approaches 33% when $n \to \infty$) as well as 5% and 25% in number of steps, compared to the approach in [17] and [16], respectively.

TABLE VI: Proposed algorithm & its steps for computing Sum and Carry-out according to Equation (5) and Equation (6)

Step	Operation	Equivalent logic
1	$s_1 = 0$	$FALSE(s_1)$
2	$s_2 = 0$	$FALSE(s_2)$
3	$a \to s_1 = s_1'$	$NOT(a)$
4	$b \to s_2 = s_2'$	$NOT(b)$
5	$s_1' \to b = b'$	$NOT(a) \to b$
6	$a \to s_2' = s_2''$	$a \to NOT(b)$
7	$a = 0$	$FALSE(a)$
8	$b' \to a = a'$	$NOT(NOT(a) \to b)$
9	$s_2'' \to a' = a''$	$NOT(XOR(a,b))$
10	$s_1 = 0$	$FALSE(s_1)$
11	$c \to s_1 = s_1'$	$NOT(c)$
12	$s_2'' \to c = c'$	$(a \to NOT(b)) \to c$
13	$a'' \to s_1' = s_1''$	$NOT(XOR(a,b)) \to NOT(c)$
14	$a = 0$	$FALSE(a)$
15	$s_1'' \to a = a'$	$NOT[NOT(XOR(a,b)) \to NOT(c)]$
16	$s_2 = 0$	$FALSE(s_2)$
17	$c' \to s_2 = s_2'$	$NOT[(a \to NOT(b)) \to c]$
18	$b' \to s_2' = s_2''$	$(NOT(a) \to b) \to NOT[(a \to NOT(b)) \to c]$
19	$b' \to c' = c''$	$(NOT(a) \to b) \to [(a \to NOT(b)) \to c]$
20	$c'' \to a' = a''$	$\{(NOT(a) \to b) \to [(a \to NOT(b)) \to c]\} \to \{NOT[NOT(XOR(a,b)) \to NOT(c)]\}$ = Sum
21	$c = 0$	$FALSE(c)$
22	$s_2'' \to c = c'$	$NOT\{(NOT(a) \to b) \to NOT[(a \to NOT(b)) \to c]\}$ = Carry-Out

IV. CONCLUSION

Since the first implementation of passive memristor in 2008, the popularity of this device has been on the rise. Their capability to serve as elements of memory as well as logic and calculation makes them further interesting. In this paper, after reviewing the latter potential of memristors, we focused on memristor-based Full-adders which use IMPLY logic. We proposed a new algorithm which outperforms previous similar works by finishing the operation 5-25% faster, whereas it decreases the number of required memristors by 30-33%.

REFERENCES

[1] G. S. Rose, "Overview: Memristive devices, circuits and systems," in *Proceedings of 2010 IEEE ISCAS*, 2010, pp. 1955–1958.

[2] R. K. Budhathoki, M. P. Sah, S. P. Adhikari, H. Kim, and L. Chua, "Composite behavior of multiple memristor circuits," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 60, no. 10, pp. 2688–2700, 2013.

[3] L. O. Chua, "The fourth element," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1920–1927, 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, 2008.

[5] G. Sharma and L. Bhargava, "Imply logic based on TiO_2 memristor model for computational circuits," in *Int. Conf. on Circuit, Power and Computing Technologies (ICCPCT)*, 2015, pp. 1–7.

[6] P. Mazumder, S.-M. Kang, and R. Waser, "Memristors: devices, models, and applications," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1911–1919, 2012.

[7] Y. Levy *et al.*, "Logic operations in memory using a memristive akers array," *Microelectronics Journal*, vol. 45, no. 11, pp. 1429–1437, 2014.

[8] C. D. Wright, P. Hosseini, and J. A. V. Diosdado, "Beyond von-Neumann computing with nanoscale phase-change memory devices," *Advanced Functional Materials*, vol. 23, no. 18, pp. 2248–2254, 2013.

[9] H. Li, M. Hu, X. Liu, M. Mao, C. Li, and S. Duan, "Emerging memristor technology enabled next generation cortical processor," in *27th IEEE Int. System-on-Chip Conf. (SOCC)*, 2014, pp. 377–382.

[10] E. Linn, R. Rosezin, S. Tappertzhofen, R. Waser *et al.*, "Beyond von Neumann logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, p. 305205, 2012.

[11] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures*. IEEE Computer Society, 2009, pp. 33–36.

[12] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-based imply logic design procedure," in *IEEE 29th Int. Conf. on Computer Design (ICCD)*. IEEE, 2011, pp. 142–147.

[13] X. Wang *et al.*, "Memristor-based XOR gate for full adder," in *35th Chinese Control Conference (CCC)*, 2016, pp. 5847–5851.

[14] B. K'A and E. E. Swartzlander, "Memristor-based arithmetic," in *Conf. Record of the 44th Asilomar Conf. on Sig., Sys. and Comp.*, 2010.

[15] E. Lehtonen *et al.*, "Two memristors suffice to compute all boolean functions," *Electronics letters*, vol. 46, no. 3, p. 230, 2010.

[16] S. Kvatinsky *et al.*, "Memristor-based material implication (imply) logic: design principles and methodologies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, 2014.

[17] M. Teimoory, A. Amirsoleimani, J. Shamsi, A. Ahmadi, S. Alirezaee, and M. Ahmadi, "Optimized implementation of memristor-based full adder by material implication logic," in *21st IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*, 2014, pp. 562–565.