On Synthesizing Memristor-Based Logic Circuits With Minimal Operational Pulses

Hsin-Pei Wang, Chia-Chun Lin, Chia-Cheng Wu, Yung-Chih Chen, and Chun-Yao Wang, *Member, IEEE*

***Abstract*—Memristor, which is a two-terminal nanodevice, widely used in various fields, e.g., machine learning and neuromorphic systems, has attracted much attention these years. Memristor can also be used to realize an implication logic gate and thus logic circuits. However, the fanouts in a memristor-based logic circuit have some constraints and need to be processed with special care. On the other hand, in addition to the number of memristors, the number of operational pulses is another metric to measure the quality of a memristor-based logic circuit. Hence, in this paper, we propose a synthesis algorithm to deal with the fanout problems in memristor-based logic circuits using implication logic gates for having a minimal number of operational pulses. We conducted experiments on a set of MCNC benchmarks. The experimental results show that the proposed algorithm can reduce 29% operational pulses and 36% memristor count on average compared with the state-of-the-art.**

***Index Terms*—Fanout, implication logic, logic synthesis, memristor, minimization.**

I. Introduction

MEMRISTOR (a contraction for memory resistor) was named and originally proposed by Chua [5]. In addition to resistor, capacitor, and inductor, memristor is regarded as a fundamental passive circuit element. The first memristor was implemented by researchers at Hewlett-Packard Labs [23]. They demonstrated a memristive device that was based on two thin-layer titanium dioxide (TiO_2) films with a conductive doped region and an insulating undoped region as shown in Fig. 1(a).

When a voltage is applied to a TiO_2-based device, oxygen atoms in the material will diffuse and the boundary will move back and forth between the two regions. This diffusion mechanism makes the material thinner on one side and thicker on the other side. For example, if a voltage V_{app} applied to a TiO_2-based device is under a reverse bias ($>V_{open}$) as shown in Fig. 1(b), the resistance of the memristor (also called the memristance) is subject to increase. On the contrary, if a voltage V_{app} applied to a TiO_2-based device is under a forward bias ($<V_{close}$) as shown in Fig. 1(c), the memristance is subject to decrease. This indicates that a memristor has two states—high resistance and low resistance. However, when the applied voltage is removed, the memristor will stay at its resistance state and exhibit the behavior of a “memory.”

Fig. 2 is an ideal I–V characteristic of a memristor [23], [25], [31], [32]. Low (high) resistance represents the state of logic 1 (0). If an applied voltage is smaller (larger) than the threshold voltage V_{close} (V_{open}), the state of the resistance will change from high to low (low to high). Otherwise, the memristor stays at the present state. Other voltages shown in Fig. 2 are described as follows:

1) V_{set}: A threshold voltage, which is smaller than V_{close}, changes the resistance state from high to low.

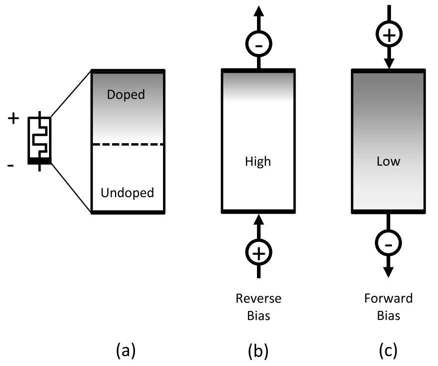


Fig. 1. (a) TiO_2-based memristor. (b) High-resistance state. (c) Low-resistance state.

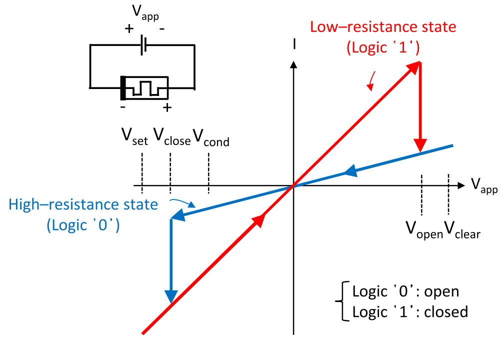


Fig. 2. Ideal I–V characteristic of a memristor.

Manuscript received August 31, 2017; revised January 30, 2018; accepted February 24, 2018. Date of publication March 29, 2018; date of current version November 30, 2018. This work was supported by the Ministry of Science and Technology of Taiwan under Grant MOST 106-2221-E-155-056, Grant MOST 103-2221-E-007-125-MY3, and Grant MOST 106-2221-E-007-111-MY3. *(Corresponding author: Chia-Cheng Wu.)*

H.-P. Wang, C.-C. Lin, C.-C. Wu, and C.-Y. Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: s104062561@m104.nthu.edu.tw; good-charlotte@hotmail.com; orange392817@gmail.com; wcyao@cs.nthu.edu.tw).

Y.-C. Chen is with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan 32003, Taiwan (e-mail: ycchen.cse@saturn.yzu.edu.tw).

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/TVLSI.2018.2816023

2) V_{clear}: A threshold voltage, which is larger than V_{open}, changes the resistance state from low to high.
3) V_{cond}: A conditional voltage that is necessary in the operations with two memristors.

The research and development of memristive devices have brought technology enhancement, but also posed challenges in different aspects. In the device implementation, different materials and compositions characterize different device properties [7], [16]–[18], [20], [21].

Generally, the favorable properties involve nonvolatility, smaller area, and lower power consumption. However, the choice of materials in the devices is highly dependent on the application. Some features of specific compounds might be beneficial in some context, but disadvantageous otherwise. An optimal memristive device for various technical requirements is imperative.

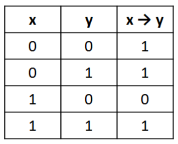
In the device modeling, a well defined and effective model of memristors is still required for a better understanding of their dynamics. Some noteworthy models in [32] and [33] were proposed by researchers.

In the logic design, memristor-based logic circuits create a new path for the exploration of new computing paradigm as well as architectures, and pose a promising alternative to conventional CMOS circuits. However, there exists various architectures and design styles that need to be evaluated and analyzed comprehensively such that a solution benefiting the most from the computing capability of memristors will be well recognized [2], [4], [5], [11], [15], [30].

Memristors have been widely used in various levels of applications, such as machine learning [3], logic circuits [2], [11], [14], [19], [27], and neuromorphic systems [28]. The applications of memristors in the logic design level, on which this paper focuses, can be classified into three categories as follows [26].

1) Memristor-only logic: Using memristors solely to implement logic operations, e.g., implication (IMPLY) logic [2], [27] and memristor-aided logic (MAGIC) [9].
2) Memristor/CMOS hybrid logic: Combining CMOS components and memristors together in Boolean logic [8], [11] and threshold logic [15].
3) Memristor-based programmable logic array: Connecting vertical and horizontal wires with memristors to form an array, such as memristor crossbar array [30].

In this paper, we use the IMPLY gate, which is a universal gate, in the first category to synthesize logic circuits. There are two metrics to measure the quality of a memristor-based logic circuit. One is the number of operational pulses, and the other is the number of memristors. Some previous works [13], [19] synthesized memristor-based circuits using only two working memristors and other input memristors. Although using only two working memristors minimizes the area of synthesized circuits, it could need a lot of pulses to operate the circuits. As a result, the synthesized circuits will have a longer delay from the viewpoint of performance. Since the size of a memristor is small, the implementation cost of a memristor is inexpensive [6], [22]. Hence, to reduce the operational delay of the synthesized circuit, we remove the constraint of using only two working memristors, and adopt more working memristors in the circuits as did in [2] and [11].



x	y	x → y
0	0	1
0	1	1
1	0	0
1	1	1

Fig. 3. Truth table of IMPLY gate.

In this paper, we propose an algorithm to synthesize memristor-based circuits for having minimal operational pulses. First of all, we synthesize a Boolean function using IMPLY gates only by a synthesis tool ABC [33]. However, the generated IMPLY network has a problem that they might be nonrealizable by memristors due to fanouts. Therefore, we have to deal with the fanout problem by restructuring the IMPLY networks in our algorithm. Furthermore, our approach also considers to synthesize a network with a minimal number of pulses during operations. We conducted experiments on a set of MCNC benchmarks. The experimental results show that the proposed algorithm can reduce 29% operational pulses and 36% memristor count on average compared with the state of the art [11].

The main contributions of this paper are threefold.

1) We propose an algorithm to synthesize memristor-based logic circuits while minimizing operational pulses.
2) We propose new structures for XOR and XNOR using IMPLY gates considering the fanout problem.
3) We solve the fanout problem in the IMPLY netlist.

The rest of this paper is organized as follows. Section II introduces the background of this paper. Section III presents our algorithm. Section IV shows the experimental results. Finally, we conclude this paper in Section V.

II. Preliminaries

A. Implication Logic

Aside from the well-known fundamental logic operations—AND, OR, and NOT—implication logic is another operation proposed [29]. It is also known as *implies* or is typically explained as *If ..., then ....* The symbol of IMPLY gate is denoted as "$\rightarrow$." For a two-input IMPLY gate "$x \rightarrow y$," its truth table is shown in Fig. 3. When x is true (1) but y is false (0), $x \rightarrow y$ is false (0). For the other cases, $x \rightarrow y$ is always true. Hence, $x \rightarrow y$ is logically equivalent to $\bar{x} + y$, and the IMPLY gate can be expressed as that shown in Fig. 4(a). An IMPLY gate can be realized by two memristors and a resistor as shown in Fig. 4(b) where an input memristor and a working memristor are connected together.

The functions of input memristor and working memristor are described as follows.

1) Input memristor: A memristor that holds an input signal and stays its resistance state after computation.
2) Working memristor: A memristor that can hold an input signal, a constant 0 value, or the operational result of an IMPLY gate.

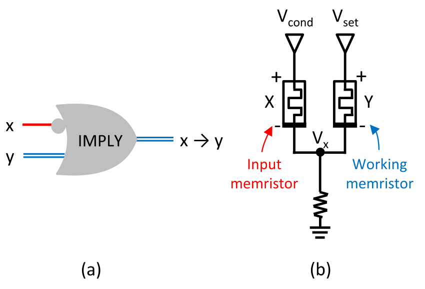


Fig. 4. (a) Symbol for IMPLY gate. (b) Memristor-realized IMPLY gate.

In Fig. 4(b), to perform the $x \rightarrow y$ operation, signals x and y need to be present on the memristors X and Y. Then, V_{cond} and V_{set} voltages are imposed on the memristors X and Y, respectively. We know that the resistance state of the memristor X will not be switched when V_{cond} is larger than V_{set} from Fig. 2. The memristor X is acted as the input memristor and connected to the negated input (single line) of the IMPLY gate in Fig. 4(a).

Next, we discuss the operations in two cases using Fig. 4(b). *Case 1:* If the memristor X is in logic "1," which means it is in the low-resistance state (closed) and V_x is approximate to V_{cond}, a voltage drop ($V_{\text{set}} - V_{\text{cond}}$) will be across the memristor Y. Hence, this voltage drop on the memristor Y is negative and larger than V_{close}, and will not cause the memristor Y change its state. The corresponding Boolean expression of this operation is as follows:

$$x = 1, \bar{x} + y = y.$$

Cases 2: If the memristor X is in logic "0," which means it is in the high-resistance state (open) and V_x is approximate to 0, a voltage drop V_{set} will be across the memristor Y. Hence, the resistance state of memristor Y will be changed from high to low according to Fig. 2. The result of this case is logic "1" and saved on the memristor Y. The corresponding Boolean expression of this operation is as follows:

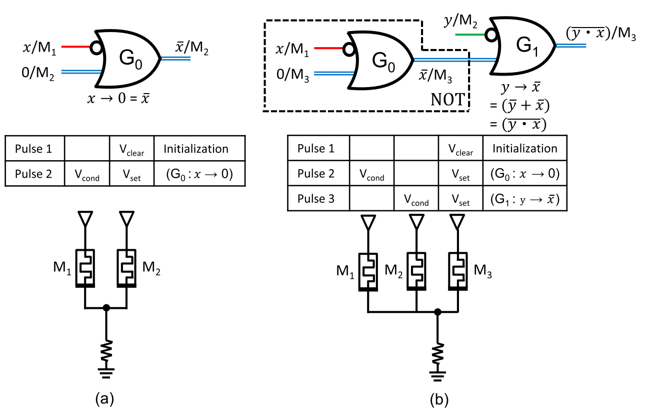
$$x = 0, \bar{x} + y = 1.$$

In summary, Fig. 4(b) realizes an IMPLY gate, and the memristor Y is acted as the working memristor connected to the nonnegated input (double lines) of the IMPLY gate in Fig. 4(a).

B. Constructing NOT and NAND Using IMPLY Gates

This section presents how to use IMPLY gates to construct NOT and NAND gates. As mentioned, the input values need to be present on the memristors before performing an IMPLY operation. Hence, when we want to construct a NOT gate, an input value x needs to be present on one memristor M_1 as shown in Fig. 5(a).[1] Also, we need to initialize the memristor M_2 as logic "0" by imposing the voltage V_{clear} on M_2, and imposing no voltage on M_1. Note that these initialization actions can be finished in the same operational pulse. Then,

[1]We use the notation x/M_1 to link the input variable x and the memristor M_1.



Pulse 1		V_{clear}	Initialization
Pulse 2	V_{cond}	V_{set}	$(G_0 : x \rightarrow 0)$

Pulse 1			V_{clear}	Initialization
Pulse 2	V_{cond}		V_{set}	$(G_0 : x \rightarrow 0)$
Pulse 3		V_{cond}	V_{set}	$(G_1 : y \rightarrow \bar{x})$

Fig. 5. (a) NOT operation. (b) NAND operation. (Blanks in the table mean no voltage applied).

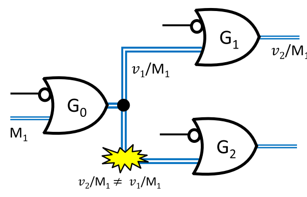


Fig. 6. Fanout problem.

the complement of x can be computed from $x \rightarrow 0$ in the next pulse, and the result $\bar{x}$ is saved on the memristor M_2. From the explanation, we know that two memristors, i.e., one input memristor M_1 and one working memristor M_2, are required to construct a NOT gate in two voltage pulses.

For a NAND gate in Fig. 5(b), two inputs x and y have to be present on the memristors M_1 and M_2, and logic "0" is set on M_3 by imposing V_{clear} and imposing no voltage on the M_1 and M_2 in Pulse 1. By performing the NOT operation as mentioned earlier, we obtain an output value $\bar{x}$, which is saved in M_3, in Pulse 2. In Pulse 3, we perform $y \rightarrow \bar{x}$, and the result $(\bar{y} + \bar{x} = \overline{y \cdot x})$ is saved in M_3. Hence, two input memristors (M_1 and M_2), one working memristor (M_3), and three operational pulses are required for a NAND gate.

C. Fanout Problem

Memristors enable *stateful* logic operations, i.e., the same memristor both performs a logic operation and saves the resultant logic value on itself [1]. Hence, there exists an inherent problem regarding the fanout of a gate in the memristor-based circuits. That is, when the output of an IMPLY gate connects to many nonnegated inputs of IMPLY gates, this output value, say v_1, intends to be used for these inputs of IMPLY gates. However, v_1 will be updated as v_2 on the same working memristor, say M_1, after computation such that the original value v_1 is lost and cannot be reused for other inputs.

For example, in Fig. 6, the output value v_1 of an IMPLY gate G_0 is saved on the working memristor M_1. Since G_0 connects to G_1 and G_2, the value on the working memristor M_1 will be updated as v_2 at the output of the IMPLY gate G_1. As a result, we will use the incorrect value v_2, which should be v_1,

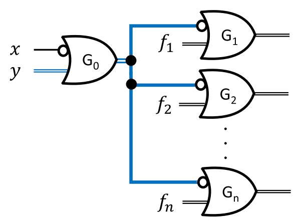


Fig. 7. Fanout connections with all input memristors.

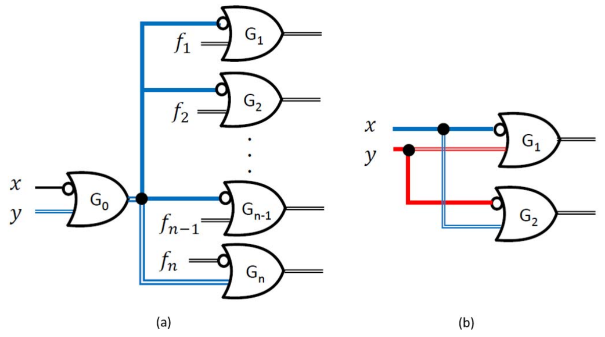


Fig. 8. Fanout connections with only one working memristor. (a) Second scenario. (b) Conflicting fanouts (third scenario).

to perform the IMPLY gate G_2 on the other fanout branch. This error is the fanout problem that needs to be dealt with in memristor-based logic circuits.

III. Algorithm

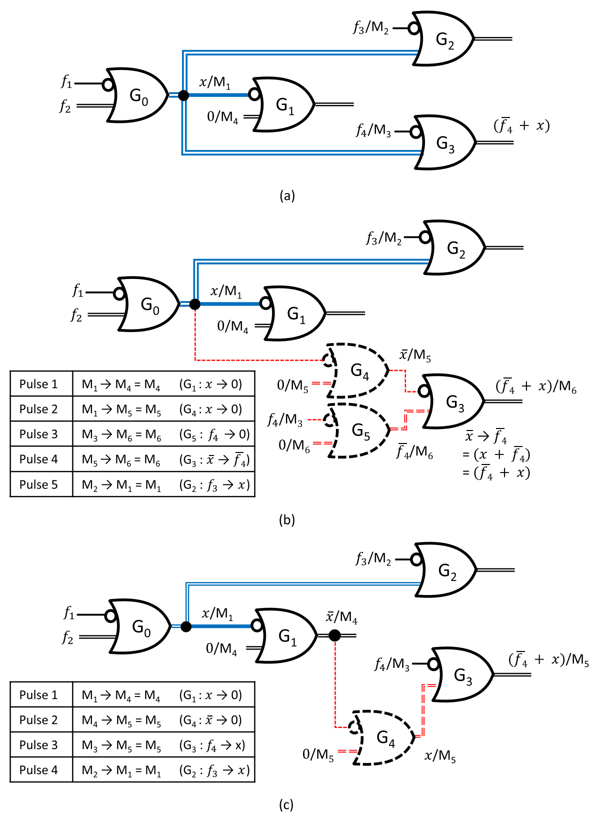
In this section, we first analyze the fanout problems in the circuits in different scenarios. Then, we present the methods for solving them. At last, we show the overall flow of the proposed algorithm.

A. Fanout Analysis

In this section, we analyze different scenarios about fanout connections, which may or may not cause incorrect results. The first scenario is that all memristors connected to the fanout are input memristors of IMPLY gates as shown in Fig. 7. In this scenario, since no fanout branch is acted as a working memristor among G_1 to G_n, the value at the output of G_0 (fanout) will not be changed and can be reused after computation on G_1 to G_n. Hence, this scenario does not incur errors.

The second scenario is that only one of the fanout branches connects to a working memristor as shown in Fig. 8(a). In this scenario, we have to consider the sequence among all operations in advance for having a correct result. Specifically, the working memristor of G_n has to work with the output value of G_0 (fanout) in the last pulse of operation sequence. If we comply with this operation sequence, the result of the circuit will be correct; otherwise, the result will be incorrect.

The third scenario is to consider two fanouts simultaneously, where one branch of each fanout is connected to one working memristor. Fig. 8(b) shows an example of such scenario. The fanout x is connected to the input memristor of G_1 and the working memristor of G_2, but y is connected to G_1 and G_2 oppositely. In this scenario, if G_1 is operated first, the output of G_1 will affect the original value y passing



Pulse 1	M$_1 \rightarrow$ M$_4$ = M$_4$ (G$_1$: $x \rightarrow 0$)
Pulse 2	M$_1 \rightarrow$ M$_5$ = M$_5$ (G$_4$: $x \rightarrow 0$)
Pulse 3	M$_3 \rightarrow$ M$_6$ = M$_6$ (G$_5$: $f_4 \rightarrow 0$)
Pulse 4	M$_5 \rightarrow$ M$_6$ = M$_6$ (G$_3$: $\bar{x} \rightarrow \bar{f}_4$)
Pulse 5	M$_2 \rightarrow$ M$_1$ = M$_1$ (G$_2$: $f_3 \rightarrow x$)

Pulse 1	M$_1 \rightarrow$ M$_4$ = M$_4$ (G$_1$: $x \rightarrow 0$)
Pulse 2	M$_4 \rightarrow$ M$_5$ = M$_5$ (G$_4$: $\bar{x} \rightarrow 0$)
Pulse 3	M$_3 \rightarrow$ M$_5$ = M$_5$ (G$_3$: $f_4 \rightarrow$ x)
Pulse 4	M$_2 \rightarrow$ M$_1$ = M$_1$ (G$_2$: $f_3 \rightarrow x$)

Fig. 9. (a) Subcircuit with the fanout problem. (b) Adding two NOT gates for the fanout problem. (c) Proposed method for the FN fanout.

to G_2. On the other hand, if G_2 is operated first, the output of G_2 will affect the original value x passing to G_1. Thus, both operation sequences will incur errors. We call this scenario as *conflicting fanouts*, which will be further discussed and solved in Section III-B.

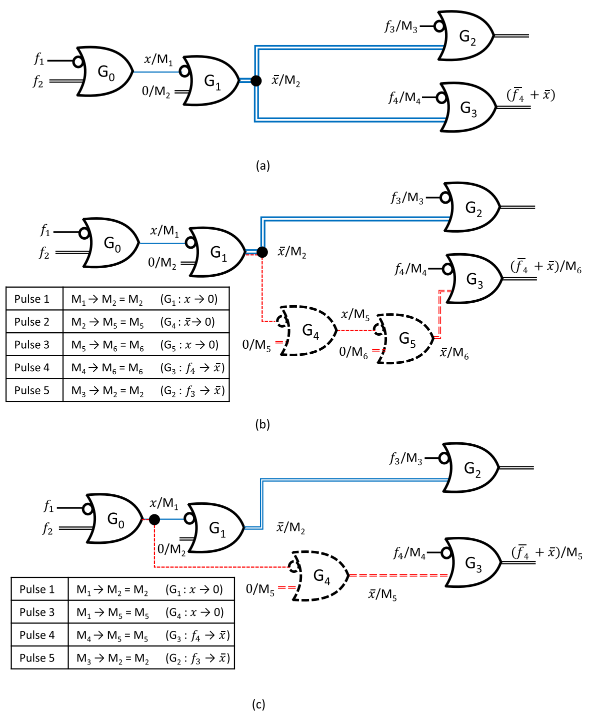
For the other scenarios that a fanout is connected to more than one working memristor, the situation is similar to the scenario mentioned in Fig. 6 and results in incorrect outputs.

B. Proposed Methods

To deal with the fanouts, except for the first and second scenarios, mentioned in Section III-A, we propose some methods in this section.

The main idea for solving the fanout problem is to duplicate the fanout value with additional working memristors in the other branches. Prior to discussing the proposed methods, we introduce two methods in [2]. The first method includes two cases. One is to copy the fanout value x by adding two NOT gates (G_4 and G_5) as shown in Fig. 9(b) where its original circuit having the fanout problem is shown in Fig. 9(a). In Fig. 9(b), the sequence of operational pulses is also listed, and the result of G_3 after Pulse 4 is $(\bar{f}_4 + x)$, which is the same as the result of G_3 in Fig. 9(a).

The other case is to recompute the fanout value x exploiting a NOT gate pair (G_4 and G_5), and use another memristor (M_6) to save the output value as shown in Fig. 10(b), where its original circuit having the fanout problem is shown in Fig. 10(a). In Fig. 10(b), the result of G_3 after Pulse 4 is $(\bar{f}_4 + \bar{x})$, which is identical to that in G_3 of Fig. 10(a).



Pulse 1	$M_1 \rightarrow M_2 = M_2$ $(G_1 : x \rightarrow 0)$
Pulse 2	$M_2 \rightarrow M_5 = M_5$ $(G_4 : \bar{x} \rightarrow 0)$
Pulse 3	$M_5 \rightarrow M_6 = M_6$ $(G_5 : x \rightarrow 0)$
Pulse 4	$M_4 \rightarrow M_6 = M_6$ $(G_3 : f_4 \rightarrow \bar{x})$
Pulse 5	$M_3 \rightarrow M_2 = M_2$ $(G_2 : f_3 \rightarrow \bar{x})$

Pulse 1	$M_1 \rightarrow M_2 = M_2$ $(G_1 : x \rightarrow 0)$
Pulse 3	$M_1 \rightarrow M_5 = M_5$ $(G_4 : x \rightarrow 0)$
Pulse 4	$M_4 \rightarrow M_5 = M_5$ $(G_3 : f_4 \rightarrow \bar{x})$
Pulse 5	$M_3 \rightarrow M_2 = M_2$ $(G_2 : f_3 \rightarrow \bar{x})$

Fig. 10. (a) Subcircuit with the fanout problem. (b) Adding two NOT gates for the fanout problem in [2]. (c) Proposed method for the BN fanout.

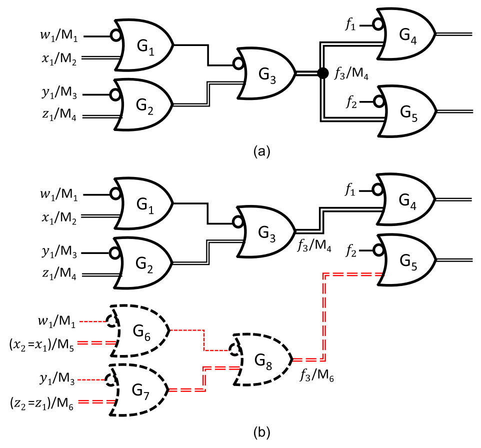


Fig. 11. Duplicating the subcircuit for the fanout problem [2].

The second method is to duplicate the whole subcircuit in the fanin cone of the fanout to obtain a copy of fanout value as shown in Fig. 11. In Fig. 11, the additional subcircuit consisting of G_6, G_7, and G_8 recomputes the fanout value of G_3 with extra operational pulses.

Although these methods solved the fanout problem successfully, they caused the number of operational pulses increase, especially for the second method. Hence, we propose three new methods to deal with the fanout problem for minimizing the operational pulse increase.

The first two new methods are related to the previous work about adding NOT gates. We use the same circuit in Figs. 9(a) and 10(a) to demonstrate the improvement of the new methods against the previous works. The first new method is for a subcircuit having a NOT gate driven by a fanout G_0 as shown in Fig. 9(a), where G_1 is a NOT gate. This fanout is called an *in front of* NOT (FN) fanout. Since the output value of G_1 is $\bar{x}$, we remove the wire connecting G_0 to the nonnegated input of G_3, and insert an additional NOT gate G_4 between G_1 and G_3 as shown in Fig. 9(c). As a result, the functionality at G_3's nonnegated input is the same as the fanout value x. Using this new method, the numbers of required memristors and pulses in Fig. 9(c) are both reduced by one compared with Fig. 9(b).

Additionally, this method is also beneficial to solve the fanout problem at the primary inputs (PIs). If an FN fanout occurs at a PI, we can solve it with this method.

Our second new method is for a subcircuit having a fanout at the output of a NOT gate G_1 as shown in Fig. 10(a). This fanout is called an *in back of* NOT (BN) fanout. In this method, we first remove one fanout branch connecting G_1 to the nonnegated input of G_3 as shown in Fig. 10(c). Since the fanout is the output of the NOT gate G_1 with the functionality $\bar{x}$, we know the functionality of negated input of the NOT gate G_1 is the complement of the fanout, x, as shown in Fig. 10(c). Hence, we can insert a NOT gate G_4 between G_0 and the original destination of the removed fanout branch. As a result, the functionality at G_3's nonnegated input is still $\bar{x}$. Using this new method, the numbers of required memristors and pulses in Fig. 10(c) are both reduced by one compared with Fig. 10(b).

The last new method is to solve the conflicting fanouts. As mentioned in Section III-A, Fig. 8(b) shows conflicting fanouts with a fanout pair (x, y). We classify the conflicting fanouts into three types based on the connectivity of (x, y) as follows.

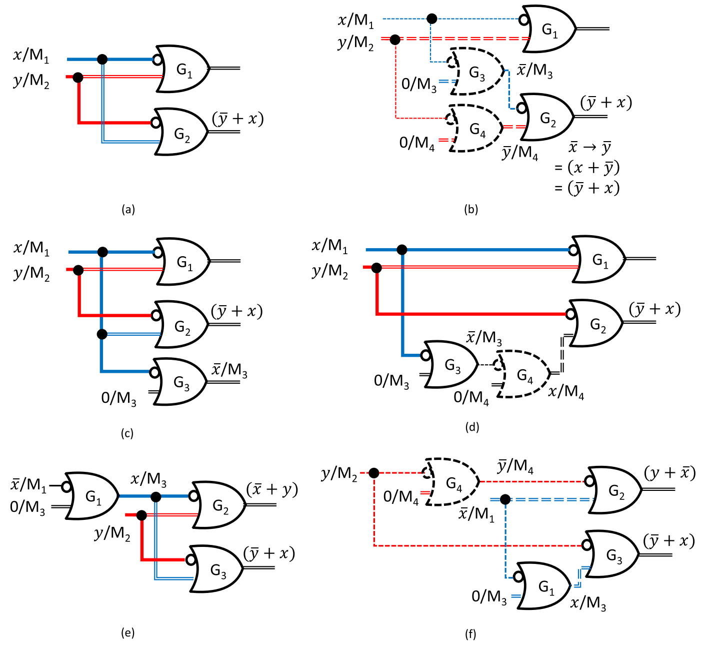
1) The fanout pair (x, y) is connected as Fig. 8(b). [We redraw it in Fig. 12(a) for convenience.] The connection with more working memristors is an extension of this type. In fact, this type only requires that x and y cannot connect to any NOT gates.
2) A branch of x (or y) is connected to a NOT gate, and only a branch of y (or x) is connected to a working memristor as shown in Fig. 12(c).
3) A fanout x (or y) is the output of a NOT gate as shown in Fig. 12(e).

For the first-type conflicting fanouts, we use the identity of $(y \rightarrow x) \equiv (\bar{x} \rightarrow \bar{y})$ to form the functionality of G_2 as shown in Fig. 12(b).

For the second-type conflicting fanouts, we apply the method for the FN fanout to solve it. As shown in Fig. 12(d), since x is connected to a NOT gate, we copy the input value x on another memristor M_4 by an additional NOT gate. Hence, the operation can be performed correctly.

For the third-type conflicting fanouts, we restructure the circuit to obtain a functionally equivalent circuit. We can see that the functionalities at G_2 and G_3 in Fig. 12(e) and (f) are equivalent.

Since XOR and XNOR gates contain conflicting fanouts, they cannot be directly realized by IMPLY gates. Thus, we have to solve the conflicting fanout problems in them. As a result, we construct new XOR and XNOR gates without having conflicting fanouts. In the following paragraphs, we elaborate the construction methods for them.

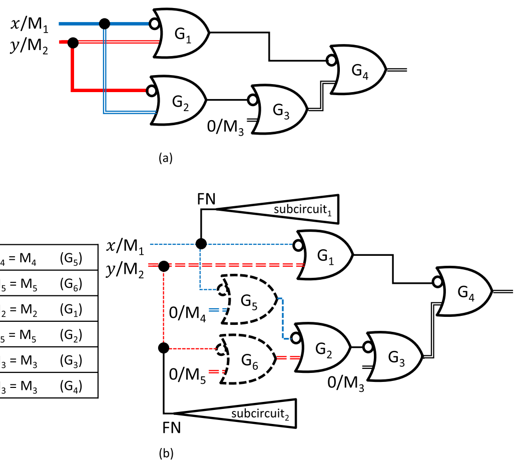


Pulse 1	$M_1 \rightarrow M_3 = M_3$ (G_3)
Pulse 2	$M_2 \rightarrow M_4 = M_4$ (G_4)
Pulse 3	$M_1 \rightarrow M_2 = M_2$ (G_1)
Pulse 4	$M_3 \rightarrow M_4 = M_4$ (G_2)

Pulse 1	$M_3 \rightarrow M_4 = M_4$ (G_4)
Pulse 2	$M_2 \rightarrow M_4 = M_4$ (G_2)
Pulse 3	$M_1 \rightarrow M_2 = M_2$ (G_1)

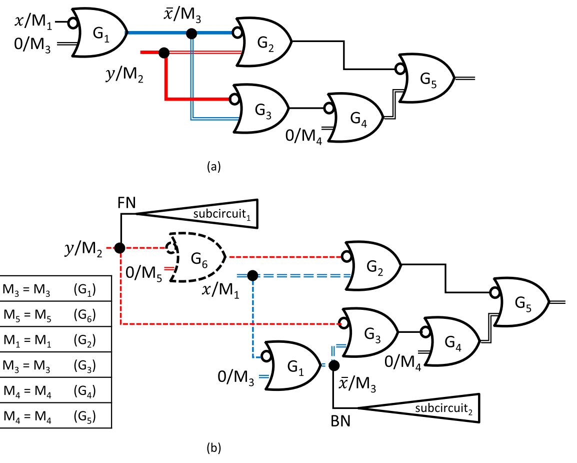
Pulse 1	$M_1 \rightarrow M_3 = M_3$ (G_1)
Pulse 2	$M_2 \rightarrow M_4 = M_4$ (G_4)
Pulse 3	$M_4 \rightarrow M_1 = M_1$ (G_2)
Pulse 4	$M_2 \rightarrow M_3 = M_3$ (G_3)

Fig. 12. (a) First-type conflicting fanouts. (b) Restructuring the subcircuit to solve the first-type conflicting fanouts. (c) Second-type conflicting fanouts. (d) Applying the method for FN fanout to solve the second-type conflicting fanouts. (e) Third-type conflicting fanouts. (f) Restructuring the subcircuit to solve the third-type conflicting fanouts.



Pulse 1	$M_1 \rightarrow M_4 = M_4$ (G_5)
Pulse 2	$M_2 \rightarrow M_5 = M_5$ (G_6)
Pulse 3	$M_1 \rightarrow M_2 = M_2$ (G_1)
Pulse 4	$M_4 \rightarrow M_5 = M_5$ (G_2)
Pulse 5	$M_5 \rightarrow M_3 = M_3$ (G_3)
Pulse 6	$M_2 \rightarrow M_3 = M_3$ (G_4)

Fig. 13. XOR gate structure. (a) With conflicting fanouts. (b) Without conflicting fanouts.



Pulse 1	$M_1 \rightarrow M_3 = M_3$ (G_1)
Pulse 2	$M_2 \rightarrow M_5 = M_5$ (G_6)
Pulse 3	$M_5 \rightarrow M_1 = M_1$ (G_2)
Pulse 4	$M_2 \rightarrow M_3 = M_3$ (G_3)
Pulse 5	$M_3 \rightarrow M_4 = M_4$ (G_4)
Pulse 6	$M_1 \rightarrow M_4 = M_4$ (G_5)

Fig. 14. XNOR gate structure. (a) With conflicting fanouts. (b) Without conflicting fanouts.

The Boolean expression of XOR using IMPLY operations is as follows:

$$
\begin{aligned}
x \oplus y &= x\bar{y} + \bar{x}y \\
&= \overline{\bar{x} + y} + \overline{\bar{y} + x} \\
&= \overline{x \rightarrow y} + ((\bar{y} + x) \rightarrow 0) \\
&= \overline{x \rightarrow y} + ((y \rightarrow x) \rightarrow 0) \\
&= (x \rightarrow y) \rightarrow ((y \rightarrow x) \rightarrow 0).
\end{aligned}
$$

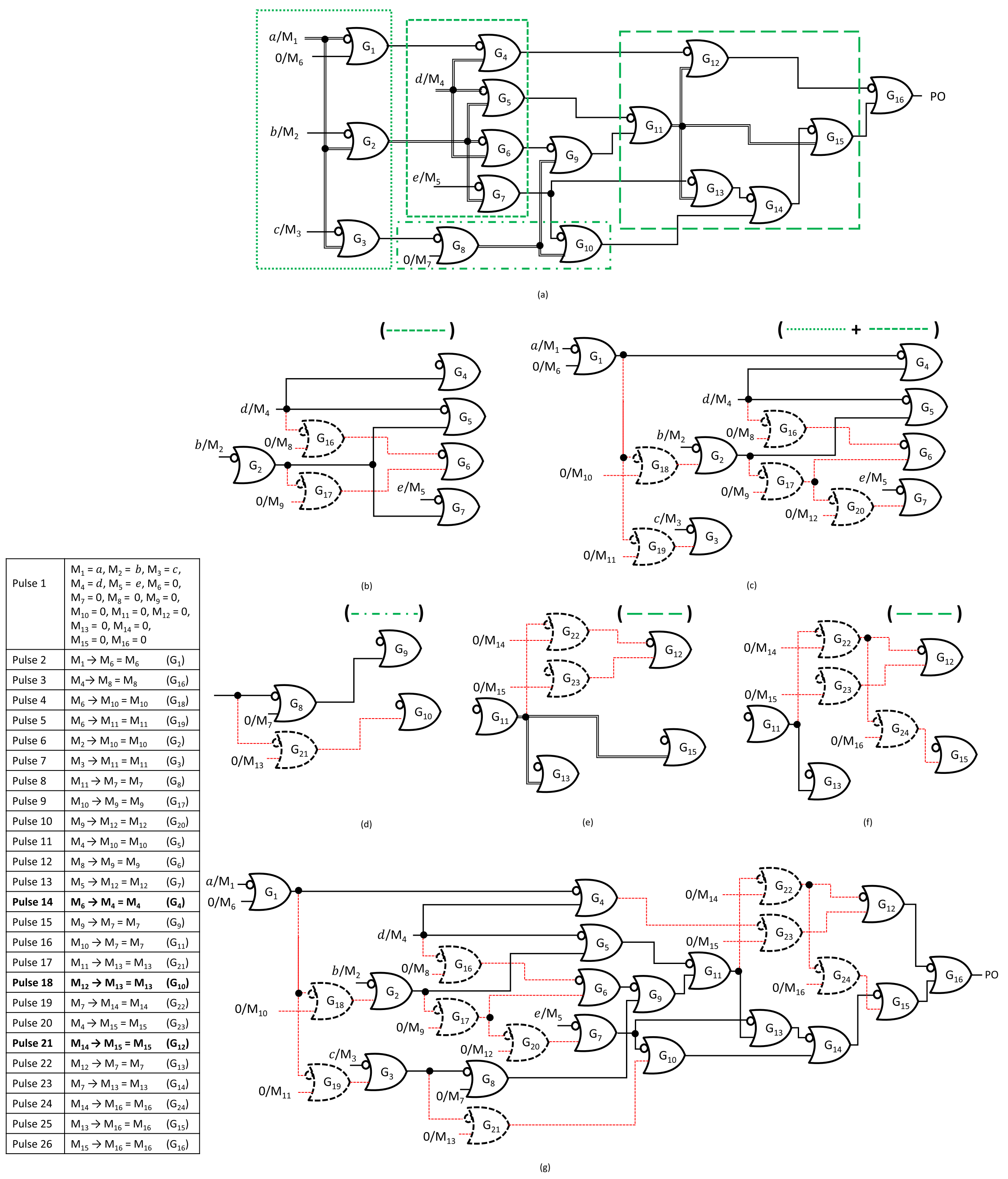
Fig. 13(a) is the corresponding IMPLY network of this expression where the fanout pair (x, y) forms conflicting fanouts. Hence, we rewrite $(y \rightarrow x)$ as $(\bar{x} \rightarrow \bar{y})$ by the law of contraposition such that

$$
\begin{aligned}
x \oplus y &= (x \rightarrow y) \rightarrow ((\bar{x} \rightarrow \bar{y}) \rightarrow 0) \\
&= (x \rightarrow y) \rightarrow (((x \rightarrow 0) \rightarrow (y \rightarrow 0)) \rightarrow 0).
\end{aligned}
$$

The resultant circuit of XOR is as shown in Fig. 13(b). The sequence of operational pulses is also listed in Fig. 13(b). Note that the added NOT gates in Fig. 13(b), i.e., G_5 and G_6, could be reused for the FN fanout in the first new method if needed. In summary, the proposed new structure for an XOR gate solves the conflicting fanout problem and only needs six operational pulses.[2]

Since XNOR is the complement of XOR, one may trivially add a NOT gate at the output of an XOR gate to construct an XNOR such that one more memristor and one more operational pulse are required. However, here we propose a novel structure for an XNOR gate without having the memristor and the operational pulse overheads.

[2]The solutions in state of the art [10] and [24] need nine and 13 operational pulses for an XOR gate, respectively.



Pulse 1	$M_1 = a$, $M_2 = b$, $M_3 = c$, $M_4 = d$, $M_5 = e$, $M_6 = 0$, $M_7 = 0$, $M_8 = 0$, $M_9 = 0$, $M_{10} = 0$, $M_{11} = 0$, $M_{12} = 0$, $M_{13} = 0$, $M_{14} = 0$, $M_{15} = 0$, $M_{16} = 0$
Pulse 2	$M_1 \rightarrow M_6 = M_6$ (G_1)
Pulse 3	$M_4 \rightarrow M_8 = M_8$ (G_{16})
Pulse 4	$M_6 \rightarrow M_{10} = M_{10}$ (G_{18})
Pulse 5	$M_6 \rightarrow M_{11} = M_{11}$ (G_{19})
Pulse 6	$M_2 \rightarrow M_{10} = M_{10}$ (G_2)
Pulse 7	$M_3 \rightarrow M_{11} = M_{11}$ (G_3)
Pulse 8	$M_{11} \rightarrow M_7 = M_7$ (G_8)
Pulse 9	$M_{10} \rightarrow M_9 = M_9$ (G_{17})
Pulse 10	$M_9 \rightarrow M_{12} = M_{12}$ (G_{20})
Pulse 11	$M_4 \rightarrow M_{10} = M_{10}$ (G_5)
Pulse 12	$M_8 \rightarrow M_9 = M_9$ (G_6)
Pulse 13	$M_5 \rightarrow M_{12} = M_{12}$ (G_7)
Pulse 14	$\mathbf{M_6 \rightarrow M_4 = M_4}$ $\mathbf{(G_4)}$
Pulse 15	$M_9 \rightarrow M_7 = M_7$ (G_9)
Pulse 16	$M_{10} \rightarrow M_7 = M_7$ (G_{11})
Pulse 17	$M_{11} \rightarrow M_{13} = M_{13}$ (G_{21})
Pulse 18	$\mathbf{M_{12} \rightarrow M_{13} = M_{13}}$ $\mathbf{(G_{10})}$
Pulse 19	$M_7 \rightarrow M_{14} = M_{14}$ (G_{22})
Pulse 20	$M_4 \rightarrow M_{15} = M_{15}$ (G_{23})
Pulse 21	$\mathbf{M_{14} \rightarrow M_{15} = M_{15}}$ $\mathbf{(G_{12})}$
Pulse 22	$M_{12} \rightarrow M_7 = M_7$ (G_{13})
Pulse 23	$M_7 \rightarrow M_{13} = M_{13}$ (G_{14})
Pulse 24	$M_{14} \rightarrow M_{16} = M_{16}$ (G_{24})
Pulse 25	$M_{13} \rightarrow M_{16} = M_{16}$ (G_{15})
Pulse 26	$M_{15} \rightarrow M_{16} = M_{16}$ (G_{16})

Fig. 15. Example. (a) Original IMPLY network with fanout problems. (b) Subcircuit after solving the conflicting fanouts. (c) Subcircuit after solving the conflicting fanouts and FN fanouts. (d) Subcircuit after solving the BN fanout. (e) Subcircuit after solving the remaining fanout. (f) Subcircuit after solving the additional FN fanout. (g) Resultant IMPLY network without fanout problems.

The Boolean expression of XNOR using IMPLY operations is as follows:

$$\begin{aligned}
\overline{x \oplus y} &= \bar{x}\bar{y} + xy \\
&= \overline{x + y} + \overline{\bar{y} + \bar{x}} \\
&= \overline{\bar{x} \rightarrow y} + ((\bar{y} + \bar{x}) \rightarrow 0) \\
&= \overline{\bar{x} \rightarrow y} + ((\bar{y} + (x \rightarrow 0)) \rightarrow 0) \\
&= \overline{\bar{x} \rightarrow y} + ((y \rightarrow (x \rightarrow 0)) \rightarrow 0) \\
&= (\bar{x} \rightarrow y) \rightarrow ((y \rightarrow (x \rightarrow 0)) \rightarrow 0) \\
&= ((x \rightarrow 0) \rightarrow y) \rightarrow ((y \rightarrow (x \rightarrow 0)) \rightarrow 0).
\end{aligned}$$

Fig. 14(a) is the corresponding IMPLY network of this expression where the fanout pair $(\bar{x}, y)$ forms conflicting fanouts. Hence, we also rewrite $((x \rightarrow 0) \rightarrow y)$ as $((y \rightarrow 0) \rightarrow x)$ such that

$$\overline{x \oplus y} = ((y \rightarrow 0) \rightarrow x) \rightarrow ((y \rightarrow (x \rightarrow 0)) \rightarrow 0).$$

The resultant circuit of XNOR is as shown in Fig. 14(b). Similarly, the NOT gates in Fig. 14(b), i.e., G_1 and G_6, could be reused in the first two new methods if needed. In summary, the proposed new structure for an XNOR gate solves the

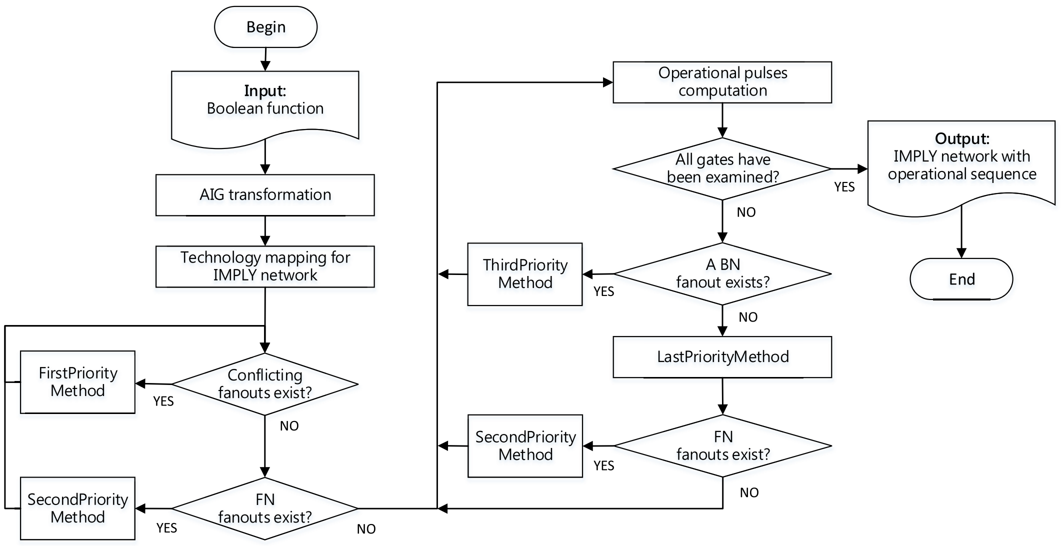


Fig. 16. Flowchart of the proposed algorithm.

conflicting fanout problem and only needs six operational pulses.

Applying these three new methods, we can solve some fanout problems in the network. However, if a fanout problem cannot be solved by the proposed new methods due to structural mismatch, we can apply the method in the previous work at last as introduced in the beginning of this section.

Next, we discuss the sequence of applying these new methods. The application of the proposed new methods will cause different effects on the resultant circuit. Hence, we prioritize the proposed new methods heuristically for achieving a minimal operational pulse increase in the algorithm as follows:

1) the third method for solving the conflicting fanouts;
2) the first method for solving the FN fanouts;
3) the second method for solving the BN fanouts;
4) the methods in the previous works for solving the remaining fanouts.

The reasons for the priorities are threefold.

1) Solving conflicting fanouts can create additional NOT gates in the network, which could be reused for FN fanouts.
2) The fanouts at the PIs are FN fanouts rather than BN fanouts.
3) The proposed methods are better than the previous work in terms of the number of operational pulses in general.

In the synthesis of memristor-based networks, in addition to the circuit structure determination, the sequence of operational pulses has to be computed as well. One can trivially divide the algorithm into two stages, i.e., solving all the fanout problems first by restructuring the circuit, then computing the sequence of operational pulses next. However, using this way, the number of operational pulses will not be minimal due to the absence of gate reuse. Thus, the determination of the circuit structure and the computation of operational pulse sequence are interleaved for different kinds of fanouts in the proposed algorithm. After our study, all the conflicting fanouts and FN fanouts can be processed first by substituting the corresponding structures. Then, we compute the sequence of pulses in the topological ordering. However, if the computation of pulse sequence encounters the other kinds of fanouts such that the circuit cannot be further processed, we iteratively solve each fanout and compute the sequence of operational pulses until the entire circuit has been processed.

For example in Fig. 15(a), there exist fanout problems in this circuit. First, conflicting fanouts at a fanout pair (d, G_2) are identified as indicated in a square. We restructure the corresponding subcircuit by adding two NOT gates G_{16} and G_{17} into it as shown in Fig. 15(b). Then, two FN fanouts at the PI a in Fig. 15(a) and at the output of G_2 in Fig. 15(b) are identified. Hence, we totally add three more NOT gates G_{18}, G_{19}, and G_{20} to solve these FN fanout problems as shown in Fig. 15(c). After restructuring the circuit, we start to compute the sequence of operational pulses. Pulse 1 is for initialization. Next, Pulse 2 to Pulse 14 are determined. Then, a BN fanout at the output of G_8 is identified and solved by adding a NOT gate G_{21} as shown in Fig. 15(d), and the corresponding sequence from Pulse 15 to Pulse 18 is determined. After that, a fanout at the output of G_{11}, which is not conflicting fanouts, FN, or BN fanout, is identified. Therefore, we solve it using the method in the previous work, i.e., adding two additional NOT gates G_{22} and G_{23}, as shown in Fig. 15(e), and the sequence from Pulse 19 to Pulse 21 is determined. Finally, an additional FN fanout, which is a byproduct of solving the fanout at the output of G_{11}, is identified and solved by adding a NOT gate G_{24} as shown in Fig. 15(f). The resultant IMPLY network without having any fanout problems is shown in Fig. 15(g), and the corresponding sequence of operational pulses is also listed. The memristors M_1 to M_5 are initialized as five PI values (a, b, c, d, and e) by applying the corresponding voltages, V_{set} or V_{clear}. The voltage V_{clear} is applied to the other memristors, M_6 to M_{16}, in the same pulse. In summary, 26 operational pulses and 16 memristors are required for performing this circuit.

```
Algorithm 1 FirstPriorityMethod:
Input: The fanout pair (x, y) for the conflicting fanouts;
       G_target ∈ the gate pair that x and y both connect to;
       G_NOT = the NOT gate that x or y connects with
Output: The restructured subcircuit without conflicting fanouts
1:  if G_NOT does not exist then          // The first type conflicting fanouts.
2:      Pin_non = the non-negated input (working memristor) of G_target;
3:      Pin_ne = the negated input (input memristor) of G_target;
4:      Wire_non = the wire that connects to Pin_non;
5:      Wire_ne = the wire that connects to Pin_ne;
6:      RemoveConnection(Wire_non, Pin_non);
7:      RemoveConnection(Wire_ne, Pin_ne);
8:      InsertNOTGate(Wire_non, Pin_ne);
9:      InsertNOTGate(Wire_ne, Pin_non);
10: end
11: elseif x or y connects to G_NOT then  // The second type conflicting fanouts.
12:     Assume x connects to G_NOT;
13:     Wire_NOT = the output of G_NOT;
14:     Pin_non ∈ the non-negated input that FanoutWire connects to;
15:     RemoveConnection(FanoutWire, Pin_non);
16:     InsertNOTGate(Wire_NOT, Pin_non);
17: end
18: elseif G_NOT connects to x or y then  // The third type conflicting fanouts.
19:     Assume x is the output of G_NOT;
20:     Pin_ne = the negated input of G_target that x connects to;
21:     Pin_non = the non-negated input of G_target that y connects to;
22:     Wire_NOTne = the wire that connects to the negated input of G_NOT;
23:     RemoveConnection(x, Pin_ne);
24:     RemoveConnection(y, Pin_non);
25:     AddConnection(Wire_NOTne, Pin_non);
26:     InsertNOTGate(y, Pin_ne);
27: end
28: return RestructuredSubcircuit;
```

Fig. 17. Pseudocode of the first priority method.

```
Algorithm 2 SecondPriorityMethod:
Input: FanoutWire = the wire for the FN fanout;
       G_NOT = the NOT gate that FanoutWire connects to;
       Wire_NOT = the output of G_NOT
Output: The restructured subcircuit without FN fanout
1: count = number of FanoutWire connected to working memristors;
2: PinWM_count = the count^th non-negated input (working memristor) that connects to;
3: while count>1 do
4:     RemoveConnection(FanoutWire, PinWM_count);
5:     InsertNOTGate(Wire_NOT, PinWM_count);
6:     count--;
7: end
8: return RestructuredSubcircuit;
```

Fig. 18. Pseudocode of the second priority method.

```
Algorithm 3 ThirdPriorityMethod:
Input: FanoutWire = the wire for the BN fanout;
       G_NOT = the NOT gate that connects to FanoutWire;
       Wire_NOTne = the wire that connects to the negated input of G_NOT
Output: The restructured subcircuit without BN fanout
1: count = number of FanoutWire connected to working memristors;
2: PinWM_count = the count^th non-negated input (working memristor) that FanoutWire connects to;
3: while count>1 do
4:     RemoveConnection(FanoutWire, PinWM_count);
5:     InsertNOTGate(Wire_NOTne, PinWM_count);
6:     count--;
7: end
8: return RestructuredSubcircuit;
```

Fig. 19. Pseudocode of the third priority method.

```
Algorithm 4 LastPriorityMethod:
Input: FanoutWire ∉ conflicting fanouts, FN fanout, BN fanout;
       Pin_non ∈ the non-negated input (working memristor) of G_target that FanoutWire connects to;
       Pin_ne = the negated input of G_target;
       Wire = the wire that connects to Pin_ne
Output: The restructured subcircuit by adding two NOT gates
1: RemoveConnection(FanoutWire, Pin_non);
2: RemoveConnection(Wire, Pin_ne);
3: InsertNOTGate(FanoutWire, Pin_ne);
4: InsertNOTGate(Wire, Pin_non);
5: return RestructuredSubcircuit;
```

Fig. 20. Pseudocode of the last priority method.

C. Overall Algorithm

Fig. 16 shows the flowchart of the proposed algorithm. Given a Boolean function, we transform it into and-inverter graph (AIG) network. Then, we map the AIG network into an IMPLY network by a technology mapping process. Next, we solve the conflicting fanouts and FN fanouts by substituting the corresponding structures followed by the operational pulse computation. After that, we deal with the remaining fanouts accompanied with the operational pulse computation iteratively. When all the fanout problems have been solved, a realizable memristor-based IMPLY network is reported.

The pseudocodes of the first priority method to the last priority method of the proposed algorithm are shown in Figs. 17–20.

IV. Experimental Results

We implemented the proposed algorithm in C++ language within an ABC environment [33]. We conducted experiments for a set of MCNC benchmarks on an Intel Xeon E5-2650V2 2.60 GHz CentOS 6.7 platform with 64-GB memory. The multioutput benchmarks are separated as many single output benchmarks in the experiments.

The experimental results are summarized in Table I. Column 1 lists the benchmark information. Column 2 shows our results including the number of required pulses and memristors. Columns 3 and 5 show the results in the state of the art [2], [11]. Columns 4 and 6 show the improvements of our approach against them. The CPU time of our approach is less than 1 s.

For example, for the rd84f1 benchmark, the approach in [2] needs 351 operational pulses, the approach in [11] needs 361 pulses, while our approach only requires 145 pulses. Our reductions against [2] and [11] are 58.69% and 59.83%, respectively. Furthermore, when considering the number of required memristors in the resultant networks, [11] needs 215 memristors while our approach only requires 67 memristors. Around two thirds of memristors are saved for synthesizing this benchmark using our approach.

According to Table I, the proposed approach only requires 47% operational pulses compared with [2] when considering all the benchmarks. The average reduction is 27.43%. As compared with [11], our approach only requires 57% pulses and 46% memristors when considering all the benchmarks. The average reductions for the number of pulses and memristors are 29.69% and 36.39%, respectively.

TABLE I

COMPARISON OF EXPERIMENTAL RESULTS AMONG OUR APPROACH AND THE STATE OF THE ART

		Ours		[2]	Pulse Redu.	[11]		Pulse Redu.	Memr. Redu.
Benchmark	|PI|	|Pulse|	|Memr.|	|Pulse|	(%)	|Pulse|	|Memr.|	(%)	(%)
exam1_d	3	13	9	12	-8.33	17	12	23.53	25.00
exam3_d	4	14	9	12	-16.67	14	8	0.00	-12.50
rd53f1	5	22	14	27	18.52	26	15	15.38	6.67
rd53f2	5	37	20	57	35.09	42	31	11.90	35.48
rd53f3	5	26	18	32	18.75	42	31	38.10	41.94
xor5_d	5	26	18	32	18.75	42	31	38.10	41.94
con1f1	7	14	12	18	22.22	15	12	6.67	0.00
con2f2	7	15	12	19	21.05	17	12	11.76	0.00
rd73f1	7	137	68	238	42.44	223	133	38.57	48.87
rd73f2	7	37	25	46	19.57	97	66	61.86	62.12
rd73f3	7	70	36	104	32.69	118	72	40.68	50.00
newill_d	8	39	23	20	-95.00	37	28	-5.41	17.86
newtag_d	8	16	14	21	23.81	17	15	5.88	6.67
rd84f1	8	145	67	351	58.69	361	215	59.83	68.84
rd84f2	8	43	29	47	8.51	117	81	63.25	64.20
rd84f3	8	10	10	23	56.52	16	16	37.50	37.50
rd84f4	8	170	79	345	50.72	363	215	53.17	63.26
9sym_d	9	575	258	1418	59.45	755	459	23.84	43.79
max46_d	9	266	121	427	37.70	369	214	27.91	43.46
sao2f1	10	61	34	102	40.20	87	55	29.89	38.18
sao2f2	10	77	41	112	31.25	104	65	25.96	36.92
sao2f3	10	188	89	380	50.53	199	122	5.53	27.05
sao2f4	10	165	77	252	34.52	202	121	18.32	36.36
sym10_d	10	488	238	1172	58.36	976	559	50.00	57.42
t481_d	16	526	251	1564	66.37	1314	805	59.97	68.82
Total	—	3180	1572	6831	—	5570	3393	—	—
Ratio	—	0.47	—	1	—	—	—	—	—
Avg. (%)	—	—	—	—	27.43	—	—	—	—
Ratio	—	0.57	0.46	—	—	1	1	—	—
Avg. (%)	—	—	—	—	—	—	—	29.69	36.39

TABLE II

EXPERIMENTAL RESULTS FOR LARGER DESIGNS

Benchmark	|PI|	|Pulse|	|Memr.|
cordicf1	23	100	69
vtx1f1	27	56	50
vtx1f2	27	62	52
vtx1f3	27	81	66
vtx1f4	27	27	35
vtx1f5	27	33	41
vtx1f6	27	110	75
x6dnf1	39	162	107
x6dnf2	39	195	126
x6dnf3	39	179	109
x6dnf4	39	159	103
x6dnf5	39	168	107
ibmf1	48	85	86
e64f1	65	88	108
soarf1	83	27	93
soarf17	83	59	111
soarf50	83	23	92

According to Table I, we observed that in addition to the operational pulse reduction, our approach can also reduce the number of required memristors for area minimization. Note that the work in the state of the art [11] is a memristor/CMOS hybrid logic design. The CMOS components in the design still occupy an extra area, which is excluded from the comparison with our work.

In Table I, we also found that our approach did not reduce the number of operational pulses compared with [2] and [11] for a certain benchmark, e.g., newill_d. The reason behind this might be that our approach conducted technology mapping first for having an initial IMPLY network, which may cause more IMPLY gates in the network compared with the previous works.

Finally, we also conducted experiments for some larger designs with more PIs to demonstrate the scalability of the proposed approach. The experimental results are listed in Table II.

From the experimental results, we realized that the proposed algorithm is structurally dependent. The number of required pulses for the benchmarks with fewer PIs might be larger than that of the benchmarks with more PIs.

V. CONCLUSION

Fanouts in memristor-based IMPLY networks are not always realizable without errors. In this paper, we analyze the fanouts in the circuits and present three methods to deal with the fanout problems with a smaller number of operational pulse increase. We also propose new XOR and XNOR structures without having fanout problems. Not only operational pulses, our approach also reduces the number of required memristors in the resultant IMPLY networks compared with the state of the art.

REFERENCES

[1] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, pp. 873–876, Apr. 2010.

[2] J. Bürger, C. Teuscher, and M. Perkowski, "Digital logic synthesis for memristors," in *Proc. Reed-Muller Workshop*, 2013, pp. 1–10.

[3] J. P. Carbajal, J. Dambre, M. Hermans, and B. Schrauwen, "Memristor models for machine learning," *Neural Comput.*, vol. 27, no. 3, pp. 725–747, 2015.

[4] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Sneak-path constraints in memristor crossbar arrays," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2013, pp. 156–160.

[5] L. O. Chua, "Memristor-the missing circuit element," *IEEE Trans. Circuit Theory*, vol. CT-18, no. 5, pp. 507–519, Sep. 1971.

[6] S. Fax, "Hewlett-packard unveils real-world memristor, chip of the future," in *Popular Science Magazine*, Apr. 2010. [Online]. Available: https://push.popsci.com/technology/article/2010-04/hewlett-packard-unveils-first-ever-memristor?dom=rss-default&src=syn

[7] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano Lett.*, vol. 10, no. 4, pp. 1297–1301, 2010.

[8] G. Kumar and K. Datta, "Design of digital functional blocks using hybrid memristor structures," in *Proc. IEEE TENCON*, Nov. 2015, pp. 1–5.

[9] S. Kvatinsky *et al.*, "MAGIC—Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[10] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "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, Oct. 2014.

[11] F. Lalchhandama, B. G. Sapui, and K. Datta, "An improved approach for the synthesis of Boolean functions using memristor based IMPLY and INVERSE-IMPLY gates," in *Proc. IEEE Annu. Symp. VLSI*, Jul. 2016, pp. 319–324.

[12] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *Proc. IEEE Int. Symp. Nanosc. Archit.*, Jul. 2009, pp. 33–36.

[13] 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, 2010.

[14] E. Lehtonen, J. Poikonen, and M. Laiho, "Implication logic synthesis methods for memristors," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2012, pp. 2441–2444.

[15] A. K. Maan, D. A. Jayadevi, and A. P. James, "A survey of memristive threshold logic circuits," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 8, pp. 1734–1746, Aug. 2017.

[16] F. Miao *et al.*, "Anatomy of a nanoscale conduction channel reveals the mechanism of a high-performance memristor," *Adv. Mater.*, vol. 23, no. 47, pp. 5633–5640, Nov. 2011.

[17] K. Miller, K. S. Nalwa, A. Bergerud, N. M. Neihart, and S. Chaudhary, "Memristive behavior in thin anodic titania," *IEEE Electron Device Lett.*, vol. 31., no. 7, pp. 737–739, Jul. 2010.

[18] A. S. Oblea, A. Timilsina, D. Moore, and K. A. Campbell, "Silver chalcogenide based memristor devices," in *Proc. IJCNN*, Jul. 2010, pp. 1–3.

[19] A. Raghuvanshi and M. Perkowski, "Logic synthesis and a generalized notation for memristor-realized material implication gates," in *Proc. IEEE ICCAD*, Nov. 2014, pp. 470–477.

[20] M. Saremi, H. J. Barnaby, A. Edwards, and M. N. Kozicki, "Analytical relationship between anion formation and carrier-trap statistics in chalcogenide glass films," *Electrochem. Lett.*, vol. 4, no. 7, pp. H29–H31, 2015.

[21] M. Saremi, S. Rajabi, H. J. Barnaby, and M. N. Kozicki, "The effects of process variation on the parametric model of the static impedance behavior of programmable metallization cell (PMC)," in *Proc. Mater. Res. Soc.*, vol. 1692, pp. 1–8, 2014.

[22] G. Satat and N. Wald, "Logic design with memristors," B.S.C Semesterial Project, Technion-Israel Inst. Technol., Haifa, Israel, Tech. Rep., 2011.

[23] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May 2008.

[24] 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 *Proc. 21st IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2014, pp. 562–565.

[25] S. Vaidyanathan and C. Volos, *Advances in Memristors, Memristive Devices and Systems*. Berlin, Germany: Springer, Mar. 2017.

[26] I. Vourkas and G. C. Sirakoulis, "Emerging memristor-based logic circuit design approaches: A review," *IEEE Circuits Syst. Mag.*, vol. 16, no. 3, pp. 15–30, 3rd Quart., 2016.

[27] X. Wang, R. Tan, and M. Perkowski, "Synthesis of memristive circuits based on stateful IMPLY gates using an evolutionary algorithm with a correction function," in *Proc. IEEE Int. Symp. Nanosc. Archit.*, Jul. 2016, pp. 97–102.

[28] S. Wang, W. Wang, C. Yakopcic, E. Shin, T. M. Taha, and G. Subramanyam, "Memristor devices for use in neuromorphic systems," in *Proc. IEEE Nat. Aerosp. Electron. Conf.*, Jul. 2016, pp. 253–257.

[29] A. N. Whitehead and B. Russell, *Principia Mathematica*, vol. 1. Cambridge, U.K.: Cambridge Univ. Press, 1910.

[30] C. Yakopcic, T. M. Taha, and R. Hasan, "Hybrid crossbar architecture for a memristor based memory," in *Proc. IEEE Nat. Aerosp. Electron. Conf.*, Jun. 2014, pp. 237–242.

[31] C. Yakopcic, T. M. Taha, G. Subramanyam, R. Pino, and S. Rogers, "Memristor SPICE modeling," in *Advances in Neuromorphic Memristor Science and Applications*. New York, NY, USA: Springer-Verlag, Jul. 2012.

[32] C. Yakopcic, T. M. Taha, G. Subramanyam, and R. E. Pino, "Generalized memristive device SPICE model and its application in circuit design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 8, pp. 1201–1214, Aug. 2013.

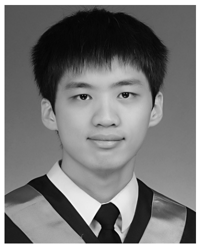
[33] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. Accessed: May 2017. [Online]. Available: https://people.eecs.berkeley.edu/~alanmi/abc/

Hsin-Pei Wang received the B.S. degree from the Department of Computer Science and Information Engineering, National Chung-Cheng University, Chiayi, Taiwan, in 2015. She is currently working toward the M.S. degree at the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan.

Her current research interests include logic synthesis, optimization, verification for VLSI designs, and automation for emerging technologies.

Chia-Chun Lin received the B.S. and M.S. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2011 and 2013, respectively, where he is currently working toward the Ph.D. degree at the Department of Computer Science.

His current research interests include logic synthesis, optimization, verification for VLSI designs, and automation for emerging technologies.



Chia-Cheng Wu received the B.S. degree from the Department of Double Specialty Program of Management and Technology, National Tsing Hua University, Hsinchu, Taiwan, in 2015, where he is currently working toward the Ph.D. degree at the Department of Computer Science.

His research interests include diagnosis and logic synthesis for emerging technologies.

Yung-Chih Chen received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2003, 2005, and 2011, respectively.

He is currently an Assistant Professor at the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan. His current research interests include logic synthesis, design verification, and design automation for emerging technologies.

Chun-Yao Wang (M'03) received the B.S. degree from the Department of Electronics Engineering, National Taipei University of Technology, Taipei, Taiwan, in 1994 and the Ph.D. degree from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 2002.

Since 2003, he has been an Assistant Professor at the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, where he is currently a Distinguished Professor. He holds nine patents. His current research interests include logic synthesis, optimization, and verification for very large-scale integrated/system-on-chip designs and emerging technologies. He has authored or coauthored over 60 technical papers in these areas.

Dr. Wang was nominated as Best Papers for two of his research results in the 2009 IEEE Asia and South Pacific Design Automation Conference and the 2010 IEEE/ACM Design Automation Conference, respectively.