Disclosing the secrets of memristors and implication logic

Jens Bürger

***Abstract*—Memristors have become interesting to many research areas within the last years. This paper discusses memristors in the context of digital logic synthesis. Giving a brief overview about memristor characteristics it is explained how they can be used in digital logic. It will be explained how memristors map to digital logic operations and how this can be used with common synthesis methods to turn an arbitrary boolean function into a logic circuit. This will highlight several fundamental differences - limitations as well as chances - between memristive and standard digital logic. The paper closes with referencing to essential publications and to address future work.**

Keywords: Memristor, IMPLY logic, material implication

I. Memristor

A. Background

Theoretically introduced was the memristor by Leon Chua in 1971 [3]. As a two terminal passive component Chua describes the memristor as the fourth basic element along resistors, capacitors and inductors. The word memristor is a composition of memory and resistor which are the two primary attributes of a memristor. After the presentation of the memristor in 1971 it found only little attention due to its missing practical implementation. This changed in 2008 when HP announced the first realization of a memristor [8]. With the physical implementation available the memristor is now subject to various research projects. This paper is structured as follows: The continuation of the first part will discuss characteristics and the functionality of memristors. In the second part implication logic as an application in digital logic is presented.

B. Functionality

As was said before a memristor has two main characteristics: memory and resistance. While a memristor is driven with a voltage that does not exceed a certain threshold it behaves resistive (not necessarily linear though). Simplified a memristor has two different resistances (often conductance is used to describe this

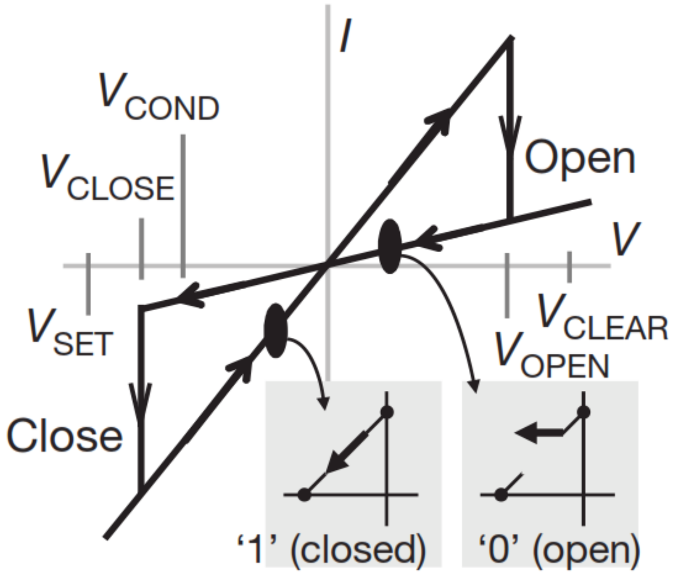

Fig. 1. Simplified IV plot of a memristor. Taken from [8]

as well). Exceeding the mentioned threshold voltage the memristor changes its resistance corresponding to the polarity of the applied voltage. The memory characteristic is due to the fact that the resistance will stay the same until the memristor is driven to the other resistance state. This implies that the resistance of the memristor can be used as binary information.

A good way to understand a memristor is to look at its IV plot. The IV plot taken from [8] is an idealized representation to better illustrate the main characteristics of memristors. In an IV plot the slope of a line describes the resistance of a component. The steeper the slope the lower the resistance as for the same applied voltage higher currents flow. In this particular case we have two linear slopes and hence two linear resistances (note that this is a simplified representation and real memristor models do not have a linear resistance). The vertical lines are the transitions between the resistances.

When in the low resistance state (steep slope) and one increases the voltage over the memristor steadily it behaves like a linear resistor until the voltage V_{Open}

V_{Open}	Voltage at which the memristor changes from low to high resistance
V_{Close}	Voltage at which the memristor changes from high to low resistance
V_{Clear}	Application specific voltage to securely switch resistance state from low to high
V_{Set}	Application specific voltage to securely switch resistance state from high to low
V_{Cond}	Voltage level necessary when performing digital operations with two memristors
closed	Memristor is in the low resistance state. This is considered as logic '1'
open	Memristor is in the high resistance state. This is considered as logic '0'
Input memristor	A memristor that holds an input signal and is not changed as a result of the computation
Working memristor	A memristor that holds either the FALSE value '0', an input value or the result of a computation.

TABLE I
MEMRISTOR NAMING CONVENTIONS

p	q	p→q
0	0	1
0	1	1
1	0	0
1	1	1

Fig. 2. IMPLY truth table

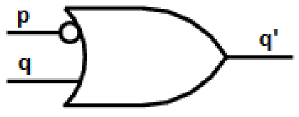

Fig. 3. Symbol of IMPLY gate

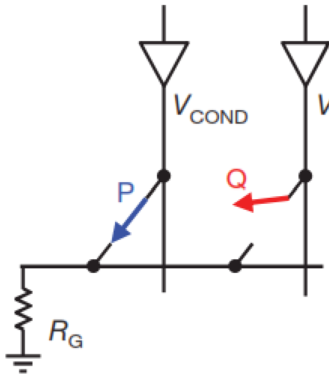

Fig. 4. IMPLY build with two memristors and one resistor and its corresponding logic symbol. Taken from [8]

is exceeded. In this moment the resistance switches from low to high. A further increase of the voltage does not have any effect. To set the memristor back to low resistance one has to apply a negative voltage greater than V_{Close}. Remember that the resistance of the memristor is a non-volatile state as long as one is within V_{Open} and V_{Close}. This means that the resistance stays the same even though the memristor is taken off from power. When applying voltage again it will show the same resistance as before. With regard to digital logic this can be mapped to logic '0' and logic '1'.

Table I introduces some naming conventions relevant to memristors and its application in digital logic.

II. IMPLICATION LOGIC

A. Background

Boolean algebra describes logic operators that based on input values '0' and '1' compute an output value. Among others, commonly known operators are AND, OR and NOT. A logic operator not so well known is material implication. The notation of material implication is p→q (spoken: p implies q) and its truth table is given in figure 2. Putting the truth table into words, if p is true than the output is equal to q. If p is not true than the output is true.

The relevance of material implication in a context of memristors was shown in [8] and will be explained in detail here as well. It was proven that material implication can be done with only 2 memristors and a single resistor. This is a highly area efficient implementation of a logic gate and hence is the motivation for exploring memristors within digital logic applications. The next paragraphs will introduce material implication based on memristors - called implication logic. When talking about gates itself the expression IMPLY gate will be used.

B. IMPLY gate

As was mentioned before the implementation of an IMPLY gate based on memristors can be done with as little as two memristors and one resistor as shown in figure 4. In this illustration the memristors are drawn as simple switches - either open or closed. This is due to the fact that in digital logic memristors are considered as two-state elements. The non-linear analog behavior between V_{Open} and V_{Close} is not relevant for the computation itself.

Discussing the functional principal of an IMPLY gate the terms defined in table I will be relevant. As can be seen in figure 4 two memristors suffice to compute the implication operation. A working memristor was defined as holding an input signal and being able to change its state and keeping the value. In this case memristor Q is a working memristor, whereas P is an input memristor. Before an IMPLY operation the memristors have to

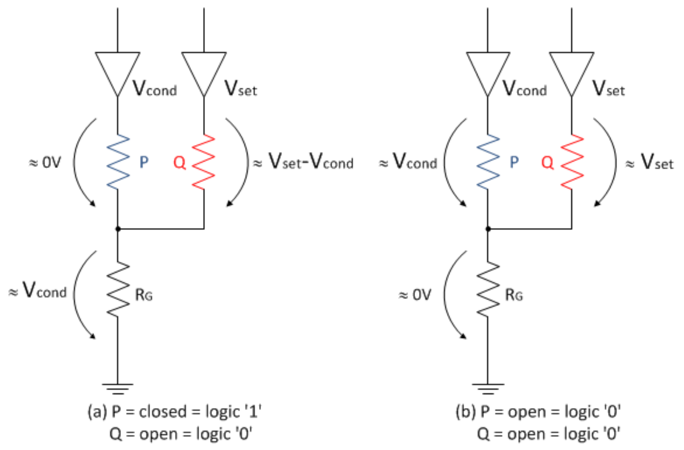

Fig. 5. Simplified IMPLY gate illustrating conditions for the workmemristor to switch to ’1’ or stay in ’0’ state. (a) voltage drop over q is V_{Set}-V_{Cond} and hence not enough to change state of q; (b) As p is open the voltage drop over q is almost V_{Set} and hence switches q to logic ’1’

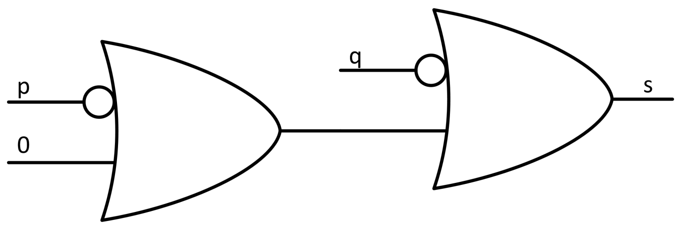

Fig. 6. NAND gate build from two IMPLY gates

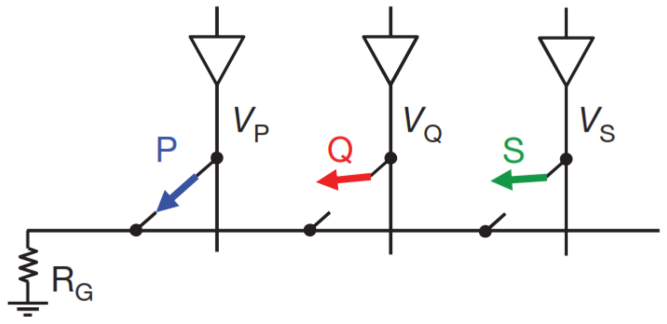

Fig. 7. NAND memristor implementation (from [8])

be set to the corresponding input values. Once both memristors are set to either ’0’ or ’1’ the computation can be performed. This is done by applying the voltage V_{Cond} to memristor P and V_{Set} to memristor Q. Looking back at figure 1 one can see that the magnitude of V_{Cond} is not sufficient to switch the memristor to another logic state. Applied to memristor P this insures that the input memristor will not change its state. V_{Set} applied to Q does have the magnitude to change the state of the memristor. But as both memristors form a voltage divider with Rg, the voltage drop over input memristor P is determining whether Q is switching or not. An illustration of that is given in figure 5.

A fact that is barely mentioned in other papers, but crucial for the understanding concerns the voltage applied to Q. As was mentioned V_{Set} and only V_{Set} is applied to Q as a working memristor. This implies that only the transition from ’0’ to ’1’ can be performed, not the transition from ’1’ to ’0’ as this would require V_{clear}. Having a look at the truth table of the IMPLY operation it becomes clear that this is sufficient. If Q = ’0’ it might change to ’1’, but with Q = ’1’ the result will be ’1’ as well.

III. Building NAND with IMPLY

While the first two sections gave an introduction to memristors and IMPLY logic it will now be discussed why this is relevant for digital logic design. IMPLY together with FALSE (setting a signal to ’0’) is a complete set of operators. This means that any arbitrary logic function can be realized using these two operations. Nowadays the most common gate in digital design and synthesis is NAND. This section will show how NAND is computed with IMPLY and give some more insights to memristive computing.

The memristive NAND operation was first shown in [8]. The logic representation of NAND build by IMPLY is shown in figure 6. This shows that a NAND can be replaced with two IMPLY gates. Out of the two memristors necessary for an IMPLY operation the non-inverted input of an IMPLY gate is computed with the working memristors. This memristor will also hold the output value and hence be the input for the next gate. This is important as a single memristor can be used for several subsequent operations. This leads to the conclusion that two IMPLY gates do not map directly to 4 memristors (as one IMPLY requires 2). In the case of the NAND as shown in figure 6 three memristors are required - two input and one working memristor.

The memristor/resistor network of a memristive NAND can be seen in figure 7. Compared to IMPLY it is only extended with one more memristor. The two inputs - p and q - are assigned to the inverted inputs of the gates (note that the inverted inputs always are the input memristors and the non-inverted input is the working memristor). The working memristor s of this implementation is used for computation for both IMPLY gates and stores the overall output result.

Reusing memristors within a logic function implies a

Step 1	Step 2			Step 3				Steps 1, 2, 3		
$s = 0$	$s' \leftarrow p\text{IMP}s$			$s'' \leftarrow q\text{IMP}s'$				$s'' \leftarrow p\overline{\text{NAND}}q$		
s	p	s	s'	q	s'	s''		p	q	s''
0	0	0	1	0	1	1	=	0	0	1
0	0	0	1	1	1	1		0	1	1
0	1	0	0	0	0	1		1	0	1
0	1	0	0	1	0	0		1	1	0

TABLE II
NAND SEQUENTIAL TRUTH TABLE (FROM [8])

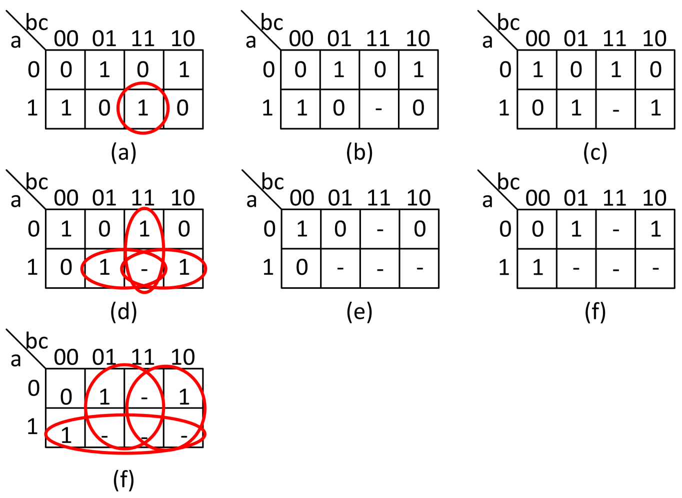

Fig. 8. (a) Prime term with positive literals (abc), (b) Replace prime term with don't care, (c) Invert, (d) Prime terms with pos. literals (ab,ac,bc), (e) Replace prime terms with don't cares, (f) Invert, (g) Prime terms with pos. literals (a,b,c)

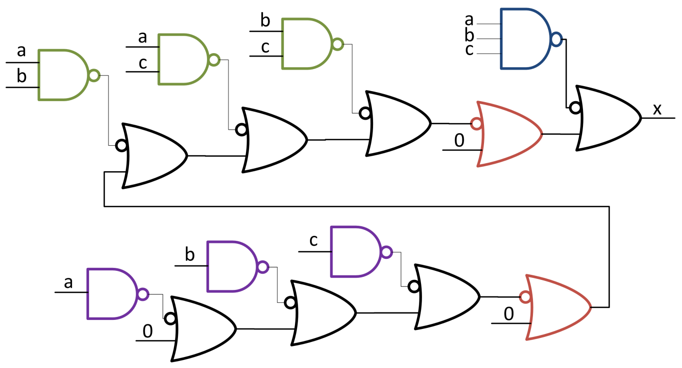

Fig. 9. Logic circuit for parity function. NAND gates will be replaced with the circuit from figure 6. The colors in the circuit correspond to the steps from figure 8 as follows: blue - (a), green - (d), purple - (g), red - (c,f). The black IMPLY gates represent the NAND form for building IMPLY circuits

sequential operation. The sequence of performing NAND with memristors is shown in table II. In a first step the FALSE operator is applied to s in order to set it to '0'. The next steps is to perform $p \rightarrow s\prime$ followed by $q \rightarrow s\prime\prime$. The need for sequential operation has basically two reasons. The first, as can be seen for the NAND gate, is that the right IMPLY gate can't compute a result without the result from the left gate. This is a classic example. But there is an IMPLY specific reason. As all memristors are connected to the same resistor they all form a single resistive network. Therefore driving additional memristors would affect the voltage levels and the functionality.

IV. Synthesis for N-Input Boolean Functions

We have seen how IMPLY gates can be used to build a 2-input NAND gate. This allows to build arbitrary circuits based on IMPLY gates. Lehtonen and Laiho proved that an arbitrary n-input boolean function can be computed with only $n + 2$ memristors [7]. This section will introduce approaches to algorithmically synthesize n-input boolean functions realized with IMPLY gates.

A. Method I - NAND form with positive literals

This first method follows an easy algorithm to step by step find prime implicants consisting of positive literals only. The algorithm searches all minterms and checks the corresponding literals. Negative literals are discarded, postive ones are kept. The resulting cube (as negative literals might be dropped, the cube is likely to get bigger) will be checked for maxterms. Due to the fact that negative literals will be dropped, often one cannot cover all minterms in a single step. In a second step all min- and maxterms are inverted and the procedure starts over again. Found minterms will be replaced with "don't cares". This will enable for creating larger cubes step by step. For illustration purpose this method is applied to the three input parity function. In figure 8 the detailed steps of this method are shown on the corresponding k-maps. Once all minterms are covered, the circuit is drawn (figure 9).

The method is described in algorithm 1. It shows how to search for prime implicants. The algorithm tries to find all minterm and checks the corresponding literals. To give an example - for a minterm $abc = 001$ the algorithm would discard a and b (as they are negative literals) and keep only c. In the next step it checks the cube c for maxterms. For the minterm $abc = 111$ the algorithm only checks this one-minterm-sized cube.

This method has created an IMPLY circuit with the

Algorithm 1 Find prime implicants with positive literals

```
while Not all minterms covered do
    while Not all minterms checked do
        Select minterm
        Keep positive literals only
        if Cube includes no maxterm then
            Save prime implicant
        end if
    end while
    Replace covered minterms with don't cares
    Invert minterms and maxterms
end while
```

depth of 2 (maximum of 2 working memristors required at a time). The advantages of this approach is the minimum number of working memristors as well as its SOP based approach. This process could be easily integrated in existing synthesis tools.

B. Method-II - including negative literals

As was described, the first method did only allow positive literals. The reason for that is to obtain the minimal number of working memristors. If one can allow for one additional working memristor prime implicants that include negative literals can be used as well. The additional memristor accounts for the negative literals that need to pass an inverter. The inverter combined with a NAND gate requires 2 working memristors for computation. The third working memristor is used to store intermediate results from other NAND gates. Figure 10 illustrates this for better understanding. As always, first the result of the lower branch of the right most IMPLY gate is computed. This result is stored in one working memristor (red line). After this b is inverted with an IMPLY and the result is stored in the second working memristor (blue line). In order to invert signal a as well one more working memristor is required (green line).

For the example of the parity function using negative literals does result in four prime implicants connected, like in Method-I, in a NAND form. Each implicant is a three input NAND gate with positive or negative inputs that would be replaced with IMPLY gates as well. In the case of the parity function it does not yield better performance (using less computaional steps). Compared to the first method with 21 steps this method requires 22 steps for the complete computation. The reason for that is the relative high number of negative literals within the prime implicants. Whereas Method-I does the inversion on a multi-input NAND, Method-II does it on each

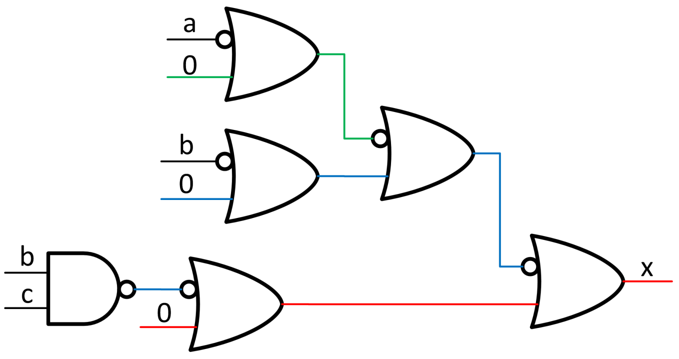

Fig. 10. This circuit shows the function $a\bar{b} + bc$. The colored lines represent the required working memristors.

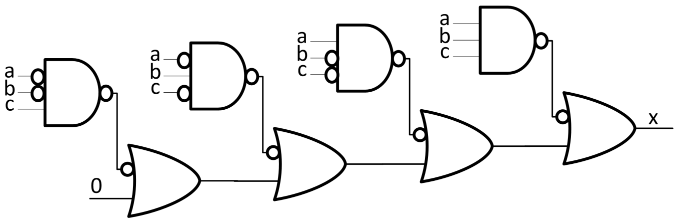

Fig. 11. Three input parity IMPLY circuit when allowing for negative literals. Note that the NAND gates need to be replaced with the corresponding IMPLY gates. Each positive input signal to a NAND gate will require a single step to compute. A negative input signal will require two steps.

negative literal. Using negative literals might result in better performance if the number of negative literals is relatively low compared to the overall number of literals. If either the one or the other method can create a better solution is function specific and can't be generalized.

C. Method-III ON/OFF list

Another method to minimize boolean expressions algorithmically is the ON/OFF list. This method compares the literals of the min- and maxterms in order to find prime implicants. This method will be demonstrated on the three input parity function to demonstrate its principle (note: as it is a SOP based method and the parity function can't be really optimized with SOP, it will not result in a better solution than the other two shown above).

For the truth table of the parity function please refer to figure 8. Figure 12 shows the ON/OFF list creation in three steps for when only positive literals are allowed. As already mentioned, this will result in a solution with the minimum number of working memristors. All minterms are written on top of columns, all maxterms at the beginning of the rows. They are then checked against each other. The literals are sorted corresponding to abc.

(a)

	001	010	100	111	
000	c	b	a	a+b+c	(a+b+c)*a*b*c = abc
011	x	x	x	a	
101				b	
110				c	

(b)

	000	011	101	110	
001	x	b	c	a+b	(b+c)*b*c = bc
010		c	a+c	a	(a+c)*a*c = ac
100		b+c	a	b	(a+b)*a*b = ab

(c)

	001	010	100		
000	c	b	a		a,b,c

Fig. 12. ON/OFF list for parity function based on positive literals only

	001	010	100	111	
000	c	b	a	a+b+c	$(\overline{a}+\overline{b}+c)*\overline{a}*\overline{b}*c = \overline{ab}c$
011	$\overline{b}$	$\overline{c}$	$a+\overline{b}+\overline{c}$	a	$(\overline{a}+b+\overline{c})*\overline{a}*b*\overline{c} = \overline{a}b\overline{c}$
101	$\overline{a}$	$\overline{a}+b+\overline{c}$	$\overline{c}$	b	$(a+\overline{b}+\overline{c})*a*\overline{b}*\overline{c} = a\overline{bc}$
110	$\overline{a}+\overline{b}+c$	$\overline{a}$	$\overline{b}$	c	

Fig. 13. ON/OFF list for parity function based on positive and negative literals

If the literal in the ON term is equal to the one in the OFF term this literal will be discarded. For a positive literal in the ON term and the corresponding literal in the OFF term being negative the positive literal will be part of the result. Vice versa (positive literal in OFF term and negative in ON term) it would generate a negative literal for the result. As these are not allowed for this example this literal will be dropped. If more than one literal is the result of a comparison they will be written in the OR form. All terms of one column will than be multiplied and the result is an (prime) implicant. If the column contained an x (no valid literal found) than this column didn't generate a prime implicant.

The restriction to only positive literals leads to the three step procedure (steps (a),(b) and (c)). After each step the min- and maxterms are inverted which leads to an additional IMPLY gate as inverter. As already mentioned, this approach created the same result as Method-I (figure 9).

When allowing for an additional work memristor one can use negative literals as well. The ON/OFF list for this is shown in figure 13. It requires only one step, but yields more complex prime implicants. However, this is the exact same solution as found for Method-II.

D. Signal generator for sequential implication logic

Another aspect relevant to IMPLY logic is the signal generator driving the memristors. As shown in figure 7 all memristors are attached to a driver. These drivers have to be controlled by a signal generator. The signal generator is executing the sequential operations by driving the corresponding memristors with either V_{Set} or V_{Cond} depending on the circuitry. Therefore the signal generator can be kept very simple. It only applies stored bit patterns in a certain interval to the output drivers.

Nevertheless a signal generator (even in it's simplest implementation) would not be worth implementing for driving a couple of memristors. The power of this technology has an effect when replicating the boolean function a hundred, a thousand or even more times. These can be driven by the same signal generator as all functions compute the same sequence. Among others, applications like computer vision, data mining or pattern recognition could greatly benefit from such an implementation.

Table III shows the signal generator pattern for the three input parity function based on positive literals. The table shows the required 21 steps and in each step one memristor is driven with V_{Set} and another with V_{Cond}. At this point the fundamental difference between standard digital logic and memristive logic becomes clear again. With standard gates the parity function can be computed with three XOR gates within 1 clock cycle. Using memristors it requires much more steps and the clock cycles for memristors will be some order of magnitude higher than for CMOS logic. Again, only a massively parallel implementation of memristors can make up for these differences.

E. Notation for IMPLY logic

Within this paper the working principle of IMPLY logic has been discussed and several examples have been shown as logic circuits. IMPLY logic differs from others in the sense that an IMPLY gate in the schematic does not map to its exclusive memristors, but rather relies on sequential execution and resource sharing. From this conceptional difference a need for a new notation for IMPLY logic might arise. This notation has to take the shared resources as well as computation time into account. Such a notation should help analyzing IMPLY logic circuits with respect to the sequential operations as well as to the required resources[1].

[1]For future work

	V_{Set}		V_{Cond}				
	W1	W2	W1	W2	A	B	C
t1	**1**	0	0	0	**1**	0	0
t2	0	**1**	**1**	0	0	0	0
t3	**1**	0	0	0	0	**1**	0
t4	0	**1**	**1**	0	0	0	0
t5	**1**	0	0	0	0	0	**1**
t6	0	**1**	**1**	0	0	0	0
t7	**1**	0	0	**1**	0	0	0
t8	0	**1**	0	0	**1**	0	0
t9	0	**1**	0	0	0	**1**	0
t10	**1**	0	0	**1**	0	0	0
t11	0	**1**	0	0	**1**	0	0
t12	0	**1**	0	0	0	0	**1**
t13	**1**	0	0	**1**	0	0	0
t14	0	**1**	0	0	0	**1**	0
t15	0	**1**	0	0	0	0	**1**
t16	**1**	0	0	**1**	0	0	0
t17	0	**1**	**1**	0	0	0	0
t18	**1**	0	0	0	**1**	0	0
t19	**1**	0	0	0	0	**1**	0
t20	**1**	0	0	0	0	0	**1**
t21	0	**1**	**1**	0	0	0	0

TABLE III
PULSE TIMING FOR MEMRISTIVE 3-INPUT PARITY FUNCTION (FIGURE 9)

V. RELATED WORK

A solid discussion of different synthesis procedures and the required number of memristors and computation step was done by Lehtonen [6]. The following paragraph will briefly summarize his ideas and results.

Contrary to [7] were it was argued that all boolean functions can be computed by using two working memristors only, [6] was actually demonstrating the trade-off between computation time and memristor count. Both, SOP and POS like methods, have been discussed and was compared at the three input parity function. By keeping the number of memristors to a minimum, both methods do not differ much in performance. Significant improvements were achieved with two different approaches.

The first one is to have complementary inputs. This means that for every input signal two memristors are required holding the complementary information. This doubles the number of input memristors, but avoids computation time for inverting signals. This approach alone can, depending on the boolean function, gain some improvement. However, in combination with the next method the best results could be achieved.

This approach was called Multi-Input Implication. When looking at the physical implementation of the memristors together with resistor R_G (see figure 5) one

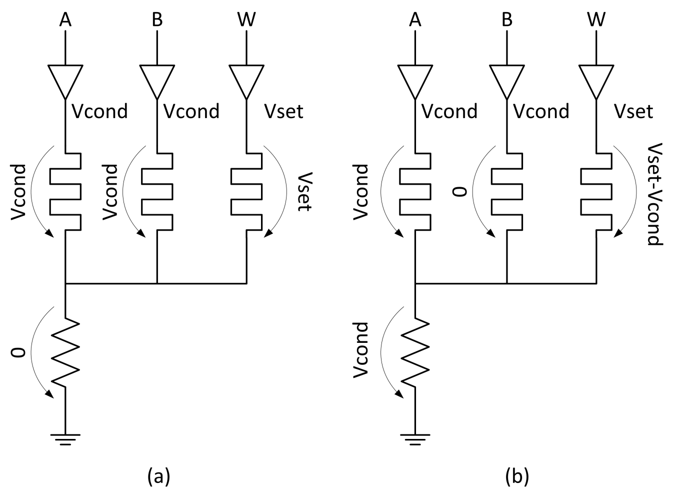

Fig. 14. Multi Input Implication - (a) both input memristors open, working memristor can switch; (b) one input memristor open, one closed. Working memristor cannot switch

can extend this to having multiple input memristors connected to this line. In case the working memristor is in the logical state '0' (high resistance), it switches to logical '1' if the input memristor is in state '0' and stays in the same state if the input is logic '1' (low resistance). Now one can think of driving the working memristor with V_{Set} and multiple input memristors with V_{Cond}. If all inputs are logic '0' (high resistance) this does not affect the voltage level over R_G and the working memristor can switch. If one or more input memristors are logic '1' the voltage level over R_G is nearly V_{Cond} and prevents the working memristor from switching. This approach maps well to OR logic as the working memristor switches if at least one input is '1' and stays '0' if and only if all inputs are '0'. (This consideration neglected the detailed discussion of the voltage drop over R_G induced by each input memristor. As parallel driven resistors/memristors will result in an overall lower resistance this would affect the ratio between memristors and R_G. But this is a design choice to make and depends on the memristor technology used)

VI. FUTURE WORK

All methods presented in this paper are based on standard logic optimization tools for SOP and POS. In [2] it was argued that simply replacing NAND and OR gates with IMPLY gates might not result in the optimal solution. A DAG based optimization routine has been presented by them to address this issue.

As Lehtonen and Laiho covered most standard optimization methods for memristive logic future work should focus on either optimizing solutions based on SOP and POS in a second step (like in [2]) or to create new methods that address the covering problem with respect to IMPLY logic in the first place. This means that one does not optimize for AND/OR and simply replace the gates, but try to map a function directly to IMPLY gates. As was shown by Lehtonen, keeping the number of input and working memristors flexible can result in significant increased performance. Therefore any future optimization algorithms should allow for a flexible number of memristors.

References

[1] Julien Borghetti, Gregory S. Snider, Philip J. Kuekes, J. Joshua Yang, Duncan R. Stewart, and R. Stanley Williams. /'memristive/' switches enable /'stateful/' logic operations via material implication. *Nature*, 464:873 –876, 4 2010.

[2] A. Chattopadhyay and Z. Rakosi. Combinational logic synthesis for material implication. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 200 –203, oct. 2011.

[3] L. Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507 – 519, sep 1971.

[4] S. Kvatinsky, A. Kolodny, U.C. Weiser, and E.G. Friedman. Memristor-based imply logic design procedure. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 142 –147, oct. 2011.

[5] E. Lehtonen and M. Laiho. Stateful implication logic with memristors. In *Nanoscale Architectures, 2009. NANOARCH '09. IEEE/ACM International Symposium on*, pages 33 –36, july 2009.

[6] E. Lehtonen, J.H. Poikonen, and M. Laiho. Two memristors suffice to compute all boolean functions. *Electronics Letters*, 46(3):239 –240, 4 2010.

[7] J.H. Poikonen, E. Lehtonen, and M. Laiho. On synthesis of boolean expressions for memristive devices using sequential implication logic. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(7):1129 –1134, july 2012.

[8] Gregory S.;Stewart Duncan R.;Williams R. Stanley Strukov, Dmitri B.;Snider. The missing memristor found. *Nature*, 453:80–83, may 2008.