Synthesis of Memristive Circuits Based on Stateful IMPLY Gates using an evolutionary algorithm with a correction function

Xiaoxiao Wang, Robin Tan, and Marek Perkowski

*Xiao University. Portland State University, Department of Electrical and Computer Engineering*

**Abstract**: Synthesis of stateful memristor-based logic circuits is realized with a multi-stage evolutionary algorithm (IMP\_ELS) which minimizes the total circuit delay. This is done by minimizing the number of pulses to control the circuit. We assume by different numbers of working memristors in the circuit and compare the delay results. The error of the synthesized circuit is the number of minterms that differ between the function of the resultant circuit and the truth table that specifies this circuit. We formulate a circuit minimization problem in which error should be zero or should be restricted to a small value. The system uses the concept of AND, OR and EXOR correction functions when the error occurs and a new round of evolution starts for the correcting function.

The logic circuit design includes coding and initialization methods to reduce random initial population of illegal and redundant solutions. Experiments with 2 to 11 input single-output functions demonstrate that the algorithm can deal with various assumed numbers of working memristors and for many benchmark functions it significantly reduces the number of pulses. The presented method can be used for both logic circuit minimization and Machine Learning in Hardware where the classifier is built as a massive logic circuit from memristors, realizing the processing in memory architecture.

1. **Introduction**

The concept of memristor was initially conceived in 1971 by Prof. Leon Chua [1], as an electronic component to link electric charge and magnetic flux. According to circuit theory, the changes in magnetic flex (ϕ) and electric charge (q) are characterized by the voltage (v) and current (i) respectively over a certain period of time (dt): dϕ=vdt and dq=idt, as shown in Figure 1. Three basic circuit components - resistor (R), inductor (L) and capacitor (C) - form a connection between two of the four fundamental elements (ϕ, q, v and i): dv=Rdi, dϕ=Ldi and dq=Cdv. It was proposed that the relationship between ϕ and q can be similarly described in a new term of memristor (M), named for memory-resistor, as dϕ=Mdq to complete the full circuit element quadrature.

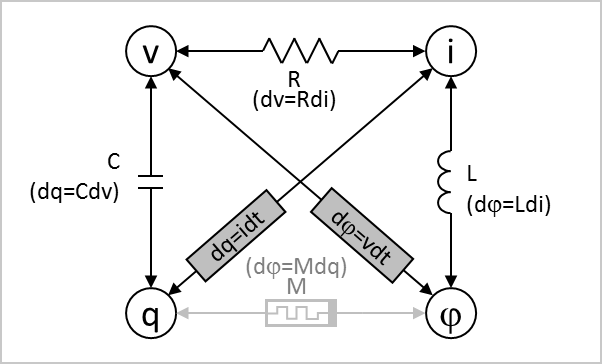


Figure 1. Circuit Components and Relationship

Memristor remained as a theoretical concept till scientists in HP Labs successfully built the first operating device in 2008 [17]. Since then, the definition of memristor has been broadened to include any type of non-volatile memory and switches, whose logic status is controlled by an excessive voltage in positive and negative directions. The application of memristors in circuit synthesis became practical after the first physical realization of a memristor based logic gate – the Material Implication (IMPLY) logic gate in 2010. The fact that using the memristor-based IMPLY gate can implement any combination of logic functions makes it possible to develop a new computing architecture where both logic operation and non-volatile data storage can take place on the same silicon substrate. It eliminates the long latency during data transferring between the computation units and data storage units found in a conventional computer. It also avoids the need for energy to transmit high speed signals over a long distance. Advanced computers based on the new architecture can deliver a superior system performance, including a much faster speed, greater computing power, more compact design, larger memory density, lower power consumption and lower system cost. It is expected that memristors will be used especially for Machine Learning and Neuromorphic architectures.

Using IMPLY gate as a basic block for reconfigurable logic synthesis has been studied in different ways. Lehtonen and his team proved in [8] that it is only necessary to use n+m+2 memristors to implement an arbitrary Boolean function f: Bn→ Bm. By representing the Boolean function in an iterative conjunctive normal form (CNF), it was further demonstrated that only n+2 memristors can build any arbitrary single-output Boolean function [9]. Using the cost heuristic strategy the team discovered an approach to further reduce the sequential length of iterative, and the number of IMPLY gates [10].

Few papers discuss the problem of synthesizing with IMPLY gates. Teodorovic et al [11] synthesize the circuit using a directed acyclic graph. Marranghello et al [12] factorize using IMPLY gates, in order to reduce the number of pulses in the circuit.

This paper presents a memristor logic based evolutionary synthesis (IMP\_ELS), aimed at obtaining the minimum total delay of the circuit for various numbers of working memristors, which correspond to circuit size. We demonstrate that using a combination of logic synthesis algorithms (SOP, ESOP) with a new type of genetic algorithm and by slightly increasing the number of working memristors one can obtain a significant reduction of total delay. The results can be used to design memristor-based FPGA-like architectures.

1. **Memristor Model and IMPLY Gate**

Figure 2 is a simplified voltage – resistance characteristics for a memristor.



Figure 2. Simplified Voltage (V) and Resistance (R) Relationship of a Memristor

When a relatively small voltage is applied to a memristor, its resistance can be low or high, illustrated as Ron and Roff in Figure 2, depending on its history. When the voltage is increased in positive direction to a level above Voff, the resistance of the memristor is changed to high, regardless its previous status. A negative voltage with an amplitude greater than that of a threshold voltage Von sets the resistance to a low state. It is important to notice that the resistance status remains no change if the voltage is between these two threshold levels of Von and Voff. The characteristics of the setting voltage – resistance hysteresis discloses the nature of memristors in memorizing its history of status. Built with advanced technologies, memristors can even reserve their status when power is lost.

To ensure a reliable operation, a margin in voltage is built in setting the status of the memristor: Vclear (>Voff) and Vset (<Von) are used to set the device to high resistance and low resistance respectively. Vcond is a bias voltage required for memristor logic operation. Figure 3 shows an equivalent circuit of a memristor.

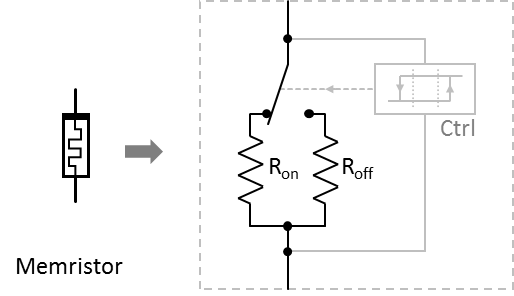


Figure 3. Symbol of Memristor (left) and Equivalent Circuit (right)

From electronic circuit point of view, a memristor can be considered as a variable resistor with two resistance levels, Ron and Roff, controlled by a decision unit (Ctrl). The selection toggles when the voltage applied to the device is above Voff or below Von, while remaining the selection during logic operation when the voltage is between the two threshold levels. It can be seen that a major differentiation of a memristor over a regular resistor is its dual resistance levels interchangeable by an excessive control voltage applied in a setting time in the range of sub nanosecond. The logic status of digital circuits using memristors is carried by the memristor resistive status, rather than a voltage.

A fundamental circuit element based on memristors is an IMPLY (memristor-based material implementation) gate. It consists of two identical memristors (MI called as input memristor, and MW called as working memristor) and one regular resistor (R), as shown in Figure 4, together with additional voltage driving circuits with input of VI and VW. The resistance of R is chosen such that Ron << R << Roff. The logic status of input and working memristors, marked as p and q, is considered as logic 1 and logic 0, if its resistance is Ron and Roff respectively.

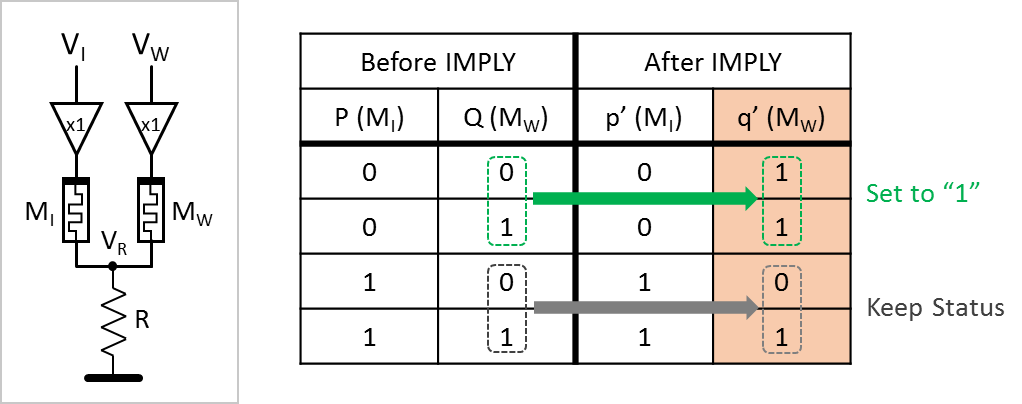


Figure 4. IMPLY Gate (left) and True Table (right)

During IMPLY logic operation, a voltage equal to Vcond is applied to MI (VI=Vcond), while the voltage at MW is set at a level equal to Vset (VM=Vset). In case of p=0, MI is at its high resistance status of Roff, VR is close to 0 since R << Roff. Thus, the voltage level on MW is Vset, which sets the resistance of MW to Ron, or p=1 according to Figure 2, regardless its previous status. When p=1, MI is at its low resistance status of Ron. Therefore, VR ≈ Vcond since Ron << R. The voltage level on MW becomes (Vset – Vcond). For properly chosen voltage levels, |Vset – Vcond|< |Vcond|, the status of MW remains no change. The true table of the IMPLY gate is shown in Figure 4, where p/q and p’/q’ present the logic status for the input/working memristors MI/MW before and after IMPLY operation respectively.

To reset the input memristor or working memristor to logic 0, it requires to hold VR to 0, while applying Vclear to VI or VM. Both memristors of an IMPLY gate can also be reset to logic 0 by one single pulse in a similar way. In this paper, the notations illustrated in Figure 5 are used, presenting one reset operation and one IMPLY gate. A Reset (Clear) operation can be also drawn as a black dot on the horizontal line representing time flow.One horizontal bar and one vertical bar present a working memristor and an input memristor respectively. Each symbol presents one pulse in time domain for logic operation.

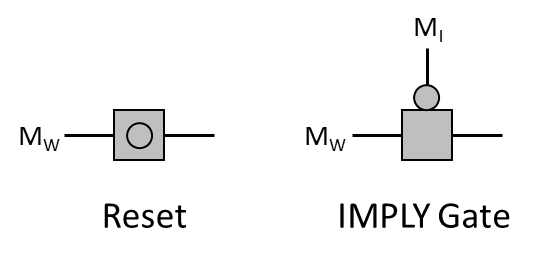


Figure 5. Reset (left) and IMPLY Gate (right) Notations Used in Genetic Analysis

1. **A combined approach to evolutionary minimization of stateful memristive circuis**

**3.1. Algorithm Description**

IMP\_ELS algorithm follows the basic framework of the general evolutionary algorithm, as shown in Table 1. The difference is that when the algorithm converges while the constraint violation has not been reduced to zero, a reminder function is calculated as a difference between the output function *o* synthesized so far and the target function *f*. The truth table of the remainder function will be used as new target function for the next run of the genetic algorithm. Detailed operation can be found in step 6 below.

Because the length of the optimal solution is unknown, the algorithm IMP\_ELS uses a chromosome with a variable length. The crossover and mutation operators as well as the population chromosome sorting algorithm are programmed similarly to [14, 15].

**3.2. Chromosome initialization method**

The maximum length of chromosome is calculated by *Lmax = n \* p + L +30*, where *n* is the number of input variables, *p* is the number of Products in SOP expression, and Lis the number of minterms in the SOP expression. The chromosome initialization length is defined as *Linitial = 2 \* Lmax / 3.*

The current state of every memristor is recorded and updated during the initialization process (Step 2 and Step 8).The state of the i-th working memristor is marked as, . During the circuit initialization or while individual memristor clearing operation, a pulse with an amplitude of *V*clear, sets this working memristor or a set of working memristors to state zero (*Si*=0). If an IMPLY gate IMP(*x*, *i*) is realized by applied pulses, and *Vx* < *I*num, then state *Si* changes from 0 to *Vx*. If Vx = Inum, then Si = Vx. If multiple consecutive IMP pulses are applied to *Si* then *Si* = *I*num+1. The purpose of recording the state of a memristor is to allow the memristor reuse. This property is unlike in classical circuits and is the main reason of a need for innovative algorithms for memistor-based logic synthesis.

**Table 1. The generic algorithm for logic circuit synthesis based on IMPLY gates using memristors**

Input: The truth table of the logic function to be synchronized.

Input parameters for GA algorithm: population size Ps, crossover probability Pc, mutation probability Pm, generation update interval U;

Output: The synthesized circuit after optimization

Step1 Calculate the maximum length of the chromosome and the length of the initial circuit.

Step 2 Initializing population P(t), t=0, U=0;

Step 3 Evaluate the initial circuit: individual coding compressing, calculating target value, and constraint violation.

Step 4 Order the sequence of chromosomes in the population.

Step 5 Record the best solution found so far.

If the obtained Best\_Solution is not updated after U generations, and the constraint violation of the Best\_Solution is greater than 0, keep the best solution in stack “Solution”.

Step 6 Calculate the truth table of the reminder function between the best function obtained Best\_Solution and the target function to be synthesized.

Delete the current population;

Step 7 Use the truth table of the reminder function as input, restart the iterations of the genetic algorithm.

Otherwise, according to Pc and Pm, select individuals for crossover and mutation, and move to new population P(t+1);

Step 8 Evaluate circuit in population P(t+1): encoding compression, calculate target value and constraint violation.

Step 9 Sort population P(t+1). Choose the Current\_Best\_Solution.

Step 10 If the Current\_Best\_Solution is better than the previous Best\_Solution, update the Best\_Solution in U; otherwise, U = U + 1;

Step 11 If the condition for out of loop is not satisfied, then t = t + 1 and go to step 7. Otherwise, take the solution from stack “Solution”, and perform circuit composition.

**Table 2. Chromosome initialization process**

Input: The number of input memristors Inum, and working memristor Wnum; The maximum chromosome length Lmax;

Output: Chromosome encoding circuit C;

Step 1 Calculate the length of chromosome initialization Linitial, Lleft = Linitial;

Step 2 Initialize output memristor state Si = 0, 

Step 3 k = rand ( )% Inum + 1;

Step 4 Randomly generated product terms to be synthesized with length k,

, and 

Step 5 Generate , write

Step 6 According to and Si, classify the set of working memristors into four categories: ZH, CH, PH, NH;

Step 7 According to PH, NH, ZH and CH priority ordering, look for new working memristor at t;

Step 8 Sequentially read from , after *m* IMP gate, generate on working memristor Wi.

Step 9 Using IMP(t, Inum+1) obtain the product term P to be synthesized;

Step 10 Lleft = Lleft-m-1;

Step 11 If Ileft <Inum, return chromosome encoding circuit C; otherwise, go to step 3.

Step 4 randomly generates small terms to be synthesized. For instance, if the small terms are *ab'd*, then P = (1, -2, 4). In Step 5, = (- 1, 2, -4).

In Step 6, the status of the i-th working memristor is recorded according to the Lj and Si values in ​. The number of all working memristors “i + Inum + 1” is stored in different sets. If Si = 0, then store in set ZH; if Si = Inum + 1, then store in set CH; if Lj > 0 and Si = Lj, store in set PH; if Lj > 0 and Si = -Lj, then store into set HH; if Lj <0, and Si = Lj, then store in set NH. Those Si not satisfying above conditions are all stored in set CH.

To obtain *ab'd,* first use working memristor *t* to generate *a '+ b + d'.* In order of priority PH, NH, ZH, CH and HH, randomly select a memristor in the first non-empty set. See step 7.

Step 8 is detailed as follows:

1. Read the Lj element in .
2. If Lj > 0, and there exist a memristor *s* in set NH whose value equals to -Lj, gate IMP(s, t) is generated; go to d;
3. If set NH does not contain memristor *s* satisfying the conditions, and if , then *s* is found, IMP(Lj,s) IMP(s,t). If, find the first non-empty set in the priority order CH, PH, NH. Randomly select a memristor *s*, and add three groups of pulses :

IMP(s, s) IMP(Lj, s) IMP(s, t).

1. Modify the states of memristors *s* and *t*;
2. j = j + 1, if j <k, go to a.

**3.3. Chromosome Compression**

Chromosome compression is necessary to remove from circuit some redundant gates after crossover and mutation operations. The following four rules are applied to determine the possibility of chromosome’s compression to eliminate the redundant gates.

**Rule 1**: If two adjacent genes in a chromosome contain the same IMPLY gate, *IMP(i,j),* then remove one of them. Because the first IMPLY gate makes the value of the memristor j as *Vj = Vi'+ Vj* , under the action of the second IMPLY gate, *Vj = Vi' + Vi '+ Vj = Vi' + Vj .*

**Rule 2**: If there are two adjacent gate in a chromosome, *IMP(i,j) and IMP(j,i),* then the second gate *IMP(j,i)* will be removed. This is because the first gate *IMP(i,j)* creates *Vj = Vi'+ Vj,* the secondgate *IMP(j, i)* makes *Vi = (Vi'+ Vj)' + Vi = ViVj'+ Vi = Vi .*

**Rule 3**: For any *IMP(i,j)* gate in a chromosome*,* if *j* is not the output memristor, and thereafter there is no *IMP(j,k) gate,* andthen delete gate *IMP(i, j) .*

**Rule 4**: If a chromosome contains two adjacent gates *IMP(i,j) and IMP(k, j)*, their position can be exchanged. This rule can be used to determine whether **Rule 1** and **Rule 2** can be applied to non-adjacent gate.

**3.4. Calculations of the remainder functions**

Let F represent the target function to be synthesized, G represent the function synthesized by the genetic algorithm, and R represent the remainder function, their corresponding truth tables can be represented as ***T***f，***T***g and ***T***r, respectively. Table 3 shows the procedure for calculating the truth table of the remainder function.

First, use the *compare( )* function to compare the relation between functions ***T***g and ***T***f. If every value in the ***Tg***truth table is equal or smaller than the corresponding value in ***Tf*** truth table, then set r = -1. If every value in the ***Tg*** truth table is equal or greater than the corresponding value in ***Tf*** truth table, then set r = +1. Otherwise, set r = 0. According to these principles, the remainder truth table is obtained by the OR, AND, or XOR operation between the target truth table ***Tf*** and the GA synthesized truth table ***Tg.***

*Table 3: Remainder function truth table calculation*

Figure 6 illustrates a simple example of the remainder function calculation, using 3-input 1-output Boolean function. *Tr* is obtainedsuch that *Tg* ∨ Tr = *Tf*. This is done using function *OrR(Tg,Tf). T*he value of each item in *Tg* and *Tf* has XOR operation, if the result obtained in Tr is 0 while the corresponding value in *Tg* is 1, then assign “don’t care” to Tr. In this example, the new remainder function to be synthesized is *a'b'c',* as presented in the middle KMap from Figure 6. Observe that since don’t cares are created in every correction process, the remainder functions obtain more and more don’t cares in next evolutionary runs, which leads to improved conditions of genetic algorithm operation and improved results.

**Input:** The truth table of the target function ***Tf***, and the truth table of the circuit synthesized function ***Tg***.

**Output:** Remainder function truth table

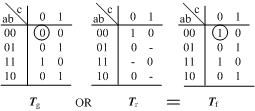
Step 1: Use function r = compare(***T***g,***T****f* ) to compare ***T***f and ***T***g

Step 2: If r = -1, ***T***r = OrR(***T***f,***T***g)

Step 3: If r = +1, ***T***r = AndR(***T***f, ***T***g)

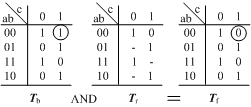
Step 4: If r = 0, ***T***r = ExorR(***T***f,***T***g)

Step 5: Exit.



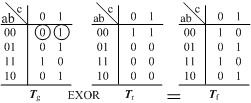
*Figure 6: OR correction: Specification function Tf is decomposed to OR operator of function Tg found so far and the remainder function Tr. The new task is now to realize function Tr. In this case some minterms are missing in function Tg so they must be added inTr.*

Figure 7 illustrates another example of remainder function calculation, where *Tr* is obtained by making *Tg* ∧ *Tr = Tf.*Function *AndR(Tg,Tf) is used to calculate the* values of each item in *Tg* and *Tf* truth table. The result is obtained for each corresponding item in *Tr* truth table. If the result is zero while the *Tg* corresponding entry is also zero, then writes '-' to the *Tr* entry. The new function to be synthesized is *a + b + c '.* Again, more don’t cares is introduced in every correction operation.



*Figure 7: AND correction: Specification function Tf is decomposed to AND operator of function Tg found so far and the remainder function Tr. The new task is now to realize function Tr. In this case some minterms are superflous in function Tg so they must be subtracted by functionTr.*

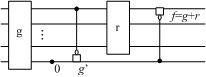
Figure 8 illustrates the method to calculate the XOR remainder function, *Tr* is obtained by making *Tg* ⊕ Tr = *Tf.* Function *EorR(Tg,Tf)* is used to calculate the value of each item in *Tg* and *Tf* truth table. The result is written in the *Tr* corresponding entries. The new function to be synthesized is *a'b'.* EXOR correction does not introduce don’t cares but rearranges the function and is also universally applicable, in contrast to the former two types of correction functions.



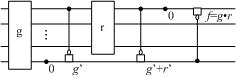
*Figure 8: EXOR correction: Specification function Tf is decomposed to EXOR operator of function Tg found so far and the remainder function Tr. The new task is now to realize function Tr.*

**3.5. Circuit Composition**

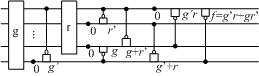
An example of using our notation representing SOP realization of function **is given in Figure 12. Using this notation to implement OR, AND and EXOR corrections is given in Figures 9, 10 and 11, respectively, for three possible relations for correcting functions; f = g ∨ r, f = g ∧ r, f= g ⊕ r. Each of them can be directly mapped to our notation of stateful memristive circuit. Horizontal lines in Figures 9, 10 and 11 represent a working memristor. For simplification, the input memristors are omitted from these figures.



*Figure 9. Structure of the OR correction in a stateful memristor circuit.*



*Figure 10. Structure of the AND correction in a stateful memristor circuit.*



*Figure 11. Structure of the EXOR correction in a stateful memristor circuit.*

Circuit blocks identified as **g** and **r** in Figures 9 - 11 represent functions *g* and *r.* The outputs of blocks **g** and **r** are in the first working memristor from the top. Other gates represent logic necessary to perform corrections. Clearing signals based on VCLEAR are denoted by a black dot with a symbol 0.Every IMPLY gate uses VCOND in the source memristor and VSET in the target working memristor.Little circle means negation and rectangledenotes an OR operator. A set of clear gates in the same column counts as one pulse and every IMPLY gate counts as one pulse.

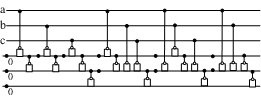
**4. Experimental results**

**4.1 Experiment settings**

The circuit optimization algorithm from [10] assumes only two working memristors for realizing all 4-input logical functions, but specific synthesis results were not presented. IMP\_ELS algorithm is designed to assume various numbers of working memristors for circuit optimization in order to minimize the number of controlling pulses. To verify and evaluate our algorithm, we tested single-output functions with 2 to 11 input variables. We calculated first the chromosome length formulas using the well-known Sum-of-Products (SOP) minimization software RONDO, and the ESOP (Exclusive-oR Sum-of-Products) minimization software EXORCISM to obtain the minimum number of pulses and compare with our method. Taking the function *Tf* from the right KMap in Figure 6 as an example, the optimized SOP expression is, and the optimized ESOP expression is 

**Method 1**: After obtaining the minimized SOP expression for a function, the circuit is realized in two modes.

In **Mode 1** the number of working memristors *Wnum* = 3. As shown in Figure 12, the first from top working memristor (just below the input memristor c) is used to create the polarity for all input variables. It should be noted that this memristor is only needed if the input has negated literals. The second working memristor from top creates products of literals, and the bottom working memristor accumulates the sum for the output SOP expression. The calculated clock pulses in this kind of implementation can be expressed by CKS=. Here *N* is the number of input variables in SOP expressions (3), *P* is the number positive variable occurrences in SOP expression (6), *T* is the number of product terms (4). *APT* is a number of terms where all variables are positive (0). Thus for our case 3\*3+6+4+0=19. 3N+P+APP =ESOP



*Figure 12. Notation used to represent function  as a SOP. Top three lines correspond to input memristors a, b, and c. Three bottom lines are ancilla bits, realized with working memristors. This circuit has 6 Clear pulses and a total of 27 pulses contributing to the total delay. Observe that the vertical arrangement of clear pulses (black dots) is counted as one pulse.*

In **Mode 2** the number of working memristors *Wnum = Inum + 2*, where *Inum* is the number of working memristors are used to obtain the negated output for each input variable. For the other two working memristors, one is used to obtain the negated value for each small item, and the other one is used to obtain the sum of final product, as shown in Figure 9. The clock pulses for this kind of circuit implementation can be expressed as:，**w**here Inum is the number of input variables (4). N=3, P=. 2T =2\*4, Inum = 3, 3+3N+P+APP =ESOP

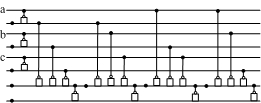
N=inp+1+Literalstotal+(prod-1)++Prod=2T+P+N

Figure 9. Circuit has 4 clear pulses. 23=4+(P=12)+(N=3)+1+(Inum=3)

**Method 2**: First step is to synthesize each product terms in ESOP minimization expression, then use *Wnum* = 5 standard module to obtain the XOR operation. To calculate the clock pulse in ESOP expression, Figure 10 shows an example of circuit synthesis for function,,, where A, B, C can be the product of any number of variables. In Figure 10, only the clock pulses on five working memristors are illustrated, while the input memristors are not shown because input memristors are only needed for the input products, and their pulse number can be obtained from method 1 above. The three modules with dashed line labeled A’, B’, and C’ are product terms, the total number of clock pulses is. The two large dash line modules implement the XOR operation, each requires 7 clock pulses. The total number of pulses for Method 2 is:3N+P+APP ESOP

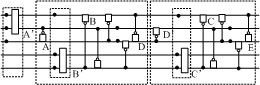


Figure 10. Circuit has 5 clear pulses. Wm = 5. 7 pulses for EXOR, T-1 = number of exors,

**4.2. Comparison of experimental results**

Using Method I, Method II and IMP\_ELS algorithm, we have tested total 22 single-output logic benchmarks with 3 to 10 input bits. The numbers of pulses for each circuit found with the statistical results are shown in Table 4. The IMP\_ELS algorithm parameters are set as follows: population size Ps = 200 for input variables <=7, and Ps = 300 for input variables >7. The probability of crossover Pc = 0.6, the probability of mutation Pm = 0.2, the generation update rate U = 400. All algorithms are running 30 times, and the best results are selected.

*Table 4. Numbers of pulses for various methods and 22 benchmark functions with from 3 to 10 inputs*

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # | Function Name | Number of inputs | # Minterms | Method1 Wnum=3 | Method1 Wnum=Imin+2 | Method2  Wnum=5 | **IMP\_ELS**  Wnum=3 | **IMP\_ELS**  Wnum=4 | **IMP\_ELS**  Wnum=5 |
| 1 | exam1\_d | 3 | 4 | 28 | 23 | 24 | 24 | 19 | ***17*** |
| 2 | exam3\_d | 4 | 6 | 33 | 25 | 31 | 29 | 21 | ***18*** |
| 3 | rd53f1 | 5 | 6 | 30 | 35 | 60 | 28 | 27 | ***26*** |
| 4 | rd53f3 | 5 | 16 | 177 | 117 | ***42*** | 160 | 137 | 130 |
| 5 | rd53f2 | 5 | 20 | 90 | 65 | 94 | 80 | 65 | ***62*** |
| 6 | con1f1 | 7 | 5 | 23 | 26 | 59 | 20 | ***17*** | ***17*** |
| 7 | con2f2 | 7 | 6 | 33 | 29 | 45 | 29 | 25 | ***24*** |
| 8 | rd73f3 | 7 | 35 | 210 | 217 | 331 | 208 | 207 | ***206*** |
| 9 | rd73f1 | 7 | 42 | 581 | 343 | ***186*** | 508 | 433 | 404 |
| 10 | rd73f2 | 7 | 64 | 961 | 583 | ***63*** | 896 | 827 | 788 |
| 11 | rd84f3 | 8 | 1 | ***10*** | 18 | ***10*** | ***10*** | ***10*** | ***10*** |
| 12 | newtag\_d | 8 | 14 | 50 | 42 | 75 | 40 | 39 | ***35*** |
| 13 | newill\_d | 8 | 22 | 82 | 65 | 124 | 76 | 64 | ***61*** |
| 14 | rd84f1 | 8 | 120 | 1288 | 764 | ***252*** | 1209 | 1022 | 1102 |
| 15 | rd84f2 | 8 | 128 | 2176 | 1288 | ***74*** | 2048 | 1915 | 1834 |
| 16 | max46\_d | 9 | 47 | 837 | ***496*** | 878 | 814 | 734 | 707 |
| 17 | 9sym\_d | 9 | 189 | 1924 | 1193 | 1118 | 1108 | 1037 | ***1035*** |
| 18 | Sao2f1 | 10 | 10 | 204 | 120 | 224 | 106 | 99 | ***94*** |
| 19 | Sao2f2 | 10 | 20 | 420 | 250 | 282 | 245 | 225 | ***208*** |
| 20 | Sao2f3 | 10 | 92 | 294 | 157 | 1338 | 149 | 139 | ***130*** |
| 21 | Sao2f4 | 10 | 85 | 235 | 139 | 267 | 95 | 93 | ***88*** |
| 22 | sym10\_d | 10 | 837 | 2310 | ***1690*** | 1916 | 2400 | 2059 | 1992 |
| 23 | 2EvenParity | 2 | 2 | 11 | 10 | 12 | 9 | 8 | ***8*** |
| 24 | 2OldParity | 2 | 2 | 10 | 10 | 11 | 9 | 8 | ***8*** |
| 25 | 3OldParity | 3 | 4 | 29 | 23 | 22 | 26 | 20 | ***19*** |
| 26 | 4EvenParity | 4 | 8 | 73 | 52 | ***32*** | 66 | 52 | 47 |
| 27 | 4OldParity | 4 | 8 | 71 | 52 | ***32*** | 61 | 54 | 46 |
| 27 | 6Mux | 6 | 4 | 25 | 26 | 42 | ***22*** | 18 | 17 |
| 28 | 11Mux | 11 | 8 | 65 | 59 | 105 | ***57*** | 46 | 43 |

**Performance Comparison:**

When comparing IMP\_ELS algorithm with Method I, mode 1 (Wnum=3), when IMP\_ELS also uses *W*num=3, the IMP\_ELS algorithm requires less the clock pulses for 20 benchmark functions. Comparing IMP\_ELS algorithm with Method I, mode 2 (*W*num=*I*num+2), the IMP\_ELS achieves better result for 10 benchmark functions. When *W*num=4, the IMP\_ELS algorithm requires less clock pulse on 14 benchmark function when comparing to Method I, mode 2. When *W*num=5, the IMP\_ELS algorithm requires less clock pulse on 15 benchmark function when comparing to Method I, mode 2. This result demonstrates that using IMP\_ELS algorithm not only finds less circuit pulses using the same number of working memristors, but demand less circuit pulses when using less working memristors.

Two methods have a relatively small numbers of pulses on five functions (rd53f3, rd73f1, rd73f2, rd84f1 and rd84f2) on which functions ESOP expressions had less literals and terms than the SOP expressions.

IMP\_ELS algorithm tries to solve the remainder function, reduce the search space, and restart the process of evolution on this basis, thereby increasing the rate of feasible solutions to accelerate convergence. In order to validate its role in the implementation the remainder functions and re-solving were not applied. By starting the process of evolution in both cases, the algorithm tests for all functions 30 times. A better solutions were found when the remainder functions were used and the evolution process was restarted. In 30 runs for each, the algorithm found correct solutions for all 22 functions. In case that this was not done, correct solutions were found only for functions 1, 2, 3, 6, 7, and 11. For the remaining functions, even in 30 runs the number of constraint violations was not reduced to zero.

**5. Conclusions**

This paper presents a new approach to synthesize memristor-based combinational logic circuits built with stateful IMPLY gates. Our comprehensive evolutionary algorithm was designed to obtain the circuit with the reduced but not necessarily exact minimum number of working memristors and it optimizes the total number of pulses. The main ideas include: creating the remainder functions and restarting the process of evolution, solving the difficult problem of equality constraints, improving the algorithm to increase the rate of feasible solutions. A total of 28 benchmark single-output logic functions with 2 to 11 inputs were minimized with the IMP\_ELS algorithm with different numbers of working memristors. For most of the test functions the algorithm found solution circuits with reduced number of pulses, and thus reduced total circuit delays. In some cases, dramatic improvements of delay, like from 1288 to 252 were found. We found that in most cases the best results are obtained for slightly increased (up to 5) number of working memristors.

Despite good results, we believe that our method can be further improved, which is the future research direction of our team. We work on more advanced correction functions, simplification rules, search strategies and partial minimization methods.

**Literature**

[1] Chua L.O., Memristor-the missing circuit element, IEEE Transactions on Circuit Theory, 1971, 18 (5), 507-519.

[2] Strukov D.B., Snider G.S., Stewart D.R., et al, The missing memristor found, Nature, 2008, 453 (7191):. pp. 80-83.

[3] Zhang J., Sun H., Li Y., et al, AgInSbTe memristor with gradual resistance tuning, Applied Physics Letters, 2013, 102 (18). 183513.

[4] Afifi A., Ayatollahi A., Raissi F., Implementation of biologically plausible spiking neural network models on the memristor crossbar-based CMOS/nano circuits, The Proceedings of IEEE: pp. 563-566.

[5] Pershin Y.V., Di Ventra M., Solving mazes with memristors: A massively parallel approach, Physical Review E, 2011, 84 (4). 046703.

[6] Gerstner E., Memristors. A-maze-ing arrays, Nature Physics, 2011, 7 (11): pp. 837-837.

[7] Borghetti J, Snider G.S., Kuekes P.J., et al, Memristive switches enable stateful logic operations via material implication, Nature, 2010, 464 (7290). pp. 873-876.

[8] Lehtonen E., Laiho M., Stateful implication logic with memristors, The Proceedings of IEEE Computer Society: pp. 33-36.

[9] Lehtonen E., Poikonen J.H., Laiho M., Two memristors suffice to compute all Boolean functions, Electronics Letters, 2010, 46 (3). 230.

[10] Poikonen J.H., Lehtonen E., Laiho M., On Synthesis of Boolean Expressions for Memristive Devices Using Sequential Implication Logic, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2012, 31 (7). pp. 1129 -1134.

[11] Teodorovic P., Dautovic S., Malbasa V., Recursive Boolean Formula Minimization Algorithms for Implication Logic, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2013, 32 (11). pp. 1829-1833.

[12] Marranghello F.S., Callegaro V., Martins M.G.A., et al, Factored Forms for Memristive Material Implication Stateful Logic, IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2015, 5 (2):. pp. 267-278.

[13] Sangho S., Kyosun K., Kang S.M., Reconfigurable Stateful NOR Gate for Large-Scale Logic-Array Integrations, *IEEE Transactions on Circuits and Systems II*: Express Briefs, , 2011, 58 (7). pp. 442- 446.

[14] Wang X, Jiao L, Li Y, et al A Variable-Length Chromosome Evolutionary Algorithm for Reversible Circuit Synthesis, *Journal of Multiple-Valued Logic & Soft Computing*, 2015, 25 (6). pp. 643-671.

[15] Wang X., Jiao L., Li Y., Hybrid algorithm based on reversible logic synthesis subset reserved, *Huazhong University of Science and Technology*: Natural Science Edition, 2015, 43 (11): pp. 51-57, *in Chinese.*

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

[17] Williams R., “How We Found The Missing Memristor”. *IEEE Spectrum*, vol.45, n.12, pp.28-35, 2008

[18] Pickett M. D, Strukov D. B., Borghetti J.L.,Yang J.J., Snider G.S., Stewart D.R., Williams R., “Switching Dynamics in Titanium Dioxide Memristive devices”, *Journal of Applied Physics*, vol. 106, 074508, 2009