Synthesis of Incompletely Specified Logic Functions With Memristor-Realized Material Implication Gates and a New Notation to Describe Circuits From Such Gates

Anika Raghuvanshi

Davidson Fellows 2014 Application
Engineering

ABSTRACT

'Memristors' are a relatively new technology that have huge potential to realize logic circuits much more efficiently than CMOS circuits due to power and size advantages. However, today's logic tools are made for CMOS circuits and do not take into account the properties of Memristor circuits. An IMPLY gate is the basic Memristor gate, as opposed to AND/OR/NOT gates in traditional CMOS circuits. A Memristor gate can be re-used repeatedly with timing pulses. A logic synthesis tool must optimize the pulse counts and number of Memristors. I designed an algorithm to realize an arbitrary logic function using only two working Memristors. I started exploring trade-offs between the number of working Memristors and pulse count. I further designed and implemented other methods making modifications to traditional approaches such as SOP, POS and ESOP minimizations. Using decision functions, I generalize the process to optimize the pulse count. I also invented the Imply Sequence Diagram, a new notation to represent Memristor circuits with pulse counts. These generalized methods and notations will be very beneficial in the realization of Memristor based circuits.

INTRODUCTION

The three basic circuit elements of electrical engineering are the resistor, capacitor, and inductor. In 1971, Leon Chua discovered the fourth fundamental element, which he called a memristor, short for '*memory resistor'* **[Chua71].** This memristor did not attract much practical interest at this time, as it was not realized as a single physical device. Since the "re-invention" of the memristor by Hewlett-Packard Corporation in 2008, many applications have emerged **[Strukov08].** Memristors have been proposed for memory design, logic-in-memory-design, standard binary combinational logic **[Borghetti10, Lehtonen09, Lehtonen10],** multiple-valued and fuzzy logic, and analog circuits. From the point of view of system synthesis, especially exciting is the possibility of having very large memories that at the same time perform some logic calculations at low cost. This topic is directly related to this paper, as a small number of working memristors (WM) is assumed.

This paper focuses on memristor-based binary logic circuits, presenting a new approach to logic synthesis using the memristor-based IMPLY gates **[Lehtonen09, Lehtonen10].** There are

also other binary gate proposals built with memristors, but the variant discussed here is the only one that has been practically realized and is supported by Hewlett-Packard **[Borghetti10].** There also exist other types of memristors **[Lehtonen12]** besides "binary switches"; these are not of interest in this paper.

BACKGROUND ON MEMRISTORS

Memristors are new passive circuit elements with interesting non-linear, analog, and memory properties that can be used for various forms of computations.

A memristor is similar to a resistor however its resistance depends on its past state. A memristor has a memory component based on the voltage it has seen in the past, hence the name "memory resistor". If a voltage is applied and the resistance increases, then that resistance will remain the same the next time that a voltage in the same direction is applied.

Compared to other circuit elements, including transistors, circuits realized with memristors have a smaller form factor due to significantly less number of physical elements needed for any given circuit. This has the potential for higher integration. While memristors have been used mainly for memories so far, they can also be used to realize combinational logic and sequential circuits from truth tables or other specifications.

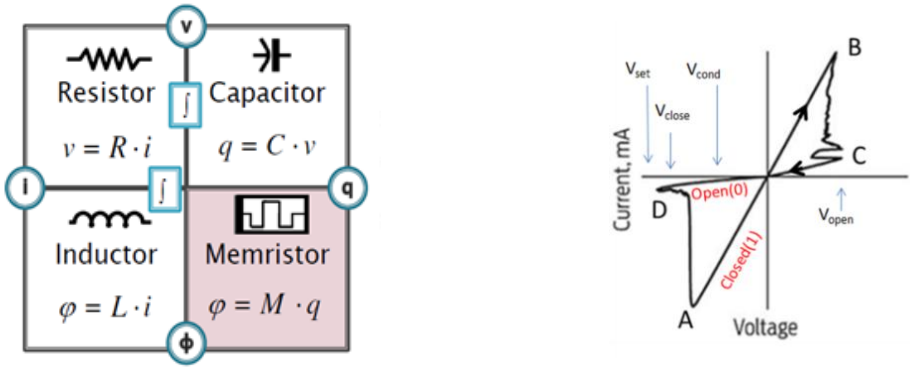


Figure 1: (a) Basic Memristor Characteristics, Source: [Strukov08],
(b) Voltage/Current Relationship

The important characteristic of a ("binary switch" type) memristor is shown in the graph in Figure 1(b), where the steep curve indicates the low resistance (the 'on' state of the memristor), as shown by line AB and the flatter curve indicates the high resistance (the 'off' state of the memristor) as shown by line interval CD.

The memristor's state given by interval AB can also be described as 'closed' or as '1' or 'true' in binary state definitions. Similarly, the state given by line interval CD can also be described as 'open' or as '0' or 'false' in binary state definitions.

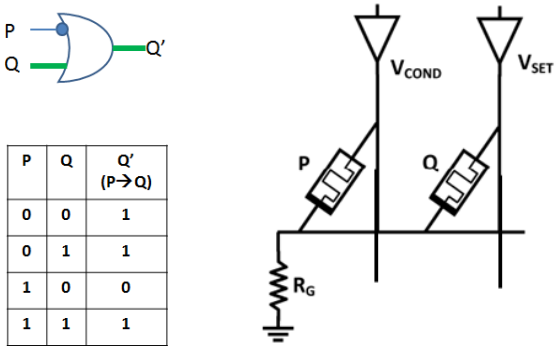
When the voltage is increased beyond a certain point, shown as V_{open}, the state of the memristor changes from closed to open (transition point B to C in the diagram). Now as the voltage is decreased and goes through the zero point, the resistance stays the same until the negative voltage exceeds V_{close}. At this point the state changes from open to closed (shown by transition from point D to A).

If the voltage remains between V_{close} and V_{open}, then there is no change in the state of the memristor.

The change of state from open to closed and closed to open allows the memristor to act as a binary switch. The fact that the state remains the same when the voltage is between V_{open} and V_{close} provides the important 'memory' property. Even when the voltage is removed, the state will remain the same, and is remembered. Observe that while a transistor is a three-terminal device, a memristor is only a two-terminal device which significantly simplifies the layout.

REALIZATION OF BASIC LOGIC GATES WITH MEMRISTORS

The IMPLY gate is the basic 'virtual' gate for memristors and it can be represented by ($P \rightarrow Q$) equivalent to ($\overline{P} + Q$), as shown in Figure 2. This is also known as a material implication, a logic gate that in contrast to NOT, AND, OR, NAND and EXOR was not used much in logic synthesis until **[Lehtonen09].** The memristor is the first technology that makes the IMPLY logic operator truly important and useful for logic synthesis.



P	Q	Q' (P→Q)
0	0	1
0	1	1
1	0	0
1	1	1

Figure 2: (a) An IMPLY Gate Truth Table,
(b) IMPLY Gate realized with two Memristors P and Q

Figure 2(a) shows the symbol and truth table for the IMPLY gate. Figure 2(b) shows the IMPLY Gate realized with two memristors. Note that this is a pulse-operated circuit, with pulses originating from voltage sources on top. Q is called the *'working memristor'* (WM for short). Q is set to either zero or one in one pulse, and two additional voltage pulses are applied to P and Q. The output is available on the same memristor Q. As shown in the diagram from Figure 2(a), Q' is the state of memristor Q after the pulse is applied.

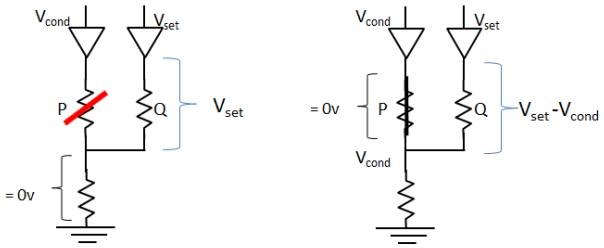


Figure 3: Workings of IMPLY gate using two Memristors.
(a) Output when P=0, (b) Output when P=1

Figure 3 shows in more details the realization of the IMPLY gate using two memristors and a grounding resistor. Figure 3(a) shows the circuit when the state of memristor P is '0' (open). P has a high resistance and can be thought of as disconnected, which implies that the voltage across grounding resistor is approximately zero. This means that the voltage across the memristor Q is very close to V_{set}. As shown in Figure 2(b), V_{set} is greater that V_{close}. The high voltage causes the state of Q to become '1' regardless of Q's original state ('0' or '1').

Figure 3(b) shows the circuit when the state of memristor P is '1'. Now P has a low resistance, and can be thought of as a wire, which implies that the voltage across the grounding resistor is

now the same as V_{cond}, the voltage applied at P input. This means that the voltage across Q is close to V_{set}-V_{cond}. Refering to Figure 1(b) again, the magnitude of V_{set}-V_{cond} is less than V_{close}, and is not enough to switch the state of Q irrespective of its previous state. This means that if Q's state was '0', it will remain '0'. If its state was '1', it will remain '1'.

Other logic gates can be realized with IMPLY gates, most easily NOT, NAND, and OR. Figure 4 presents realization of these gates, assuming that the inputs A and B already exist (not an initial situation). As shown in Figure 4, the NOT gate is simply realized by initiliazing the value of the working memristor to 0, and then applying the value of variable A to the input memristor. A point to note here is that the working memristor is always the input line going to non-negated input of the IMPLY gate, and the input memristor is the input line going to the negated input of the IMPLY gate. In the case of NOT gate as shown, the output on the working memristor would be the negated value of the input. As shown in the figure, this takes one pulse for initialization and a second pulse (a pair of pulses V_{set} and V_{cond}) for the input (this paper is not concerned with detailed generation of pulses and for simplification, the two pulses will be treated as a single pulse for simplification). The output is available after the second pulse. However, if this is a part of another circuit, the initialization can be done in parallel with a previous operation, and the NOT operation can then be done in just one additional pulse.

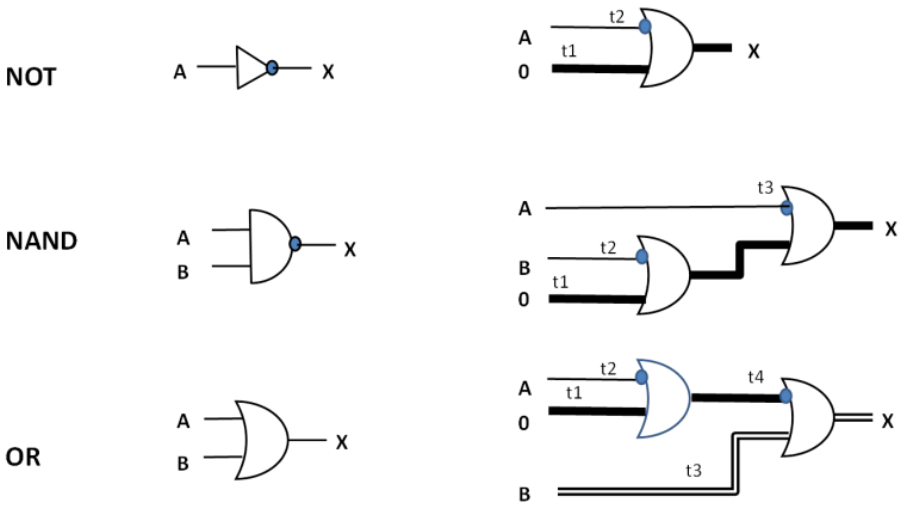


Figure 4. NOT, OR and NAND gates realized from IMPLY gates assuming that the inputs already exist.

Implementation of a two-input NAND can be done with two IMPLY gates, but just one working memristor. However, it would require 3 pulses as shown in Figure 4.

An OR gate on other hand requires two working memristors and three pulses. Thus, it can be inferred that NOT and NAND are the gates of choice for circuit synthesis rather than the OR gate. Other gates (EXOR, NOR, AND etc.) also require two working memristors and more pulses.

EXOR, XNOR, MUX, and other gates were also realized along with typical structures such as SOP, POS, negative gates, Positive Polarity Reed-Muller (PPRM) and Three level And Not networks with True Inputs (TANT) from IMPLY gates, and the trade-off between the number of WMs (see Table 2) and the total delay (number of pulses) was analyzed. the method presented here as well as other methods for synthesis with stateful IMPLY gates were created based on these analyses.

A combination of IMPLY gates, NAND gates and NOT gates will be used extensively in this paper.

MEMRISTOR VS CMOS LOGIC CIRCUITS

Memristors provide unique characteristics which make them much more efficient than currently used CMOS circuits. Because of the memory aspect of memristors, they reuse the same gate many times, therefore greatly reducing the spatial requirements. Memristor based circuits operate at significantly lower power. In CMOS circuits, if any state information needs to be persisted, the power must remain on. However, Memristors remember the state even if power is removed. This makes the overall average energy consumption of a Memristor circuit much lower than an equivalent CMOS circuit.

While Memristor circuits have benefits, the nature of the Memristor logic circuit brings some new challenges to logic synthesis. As discussed above, the basic gate in a binary memristor circuit is an Imply gate. Imply operation is not commutative as are AND/OR operations. For example, $A \rightarrow B$ is not the same as $B \rightarrow A$. So, while the sequence of operations in AND/OR expressions can be changed, the Imply operation must preserve the sequence.

Memristor gates operate by changing the state of a working memristor with every pulse. New input combines with existing state and produces the new state. This leads to a problem that a working memristor cannot be used to feed the state to more than one gate. So, a fan-out from a memristor gate is not possible, and additional steps need to be taken to address this. This is illustrated in Figure 5.

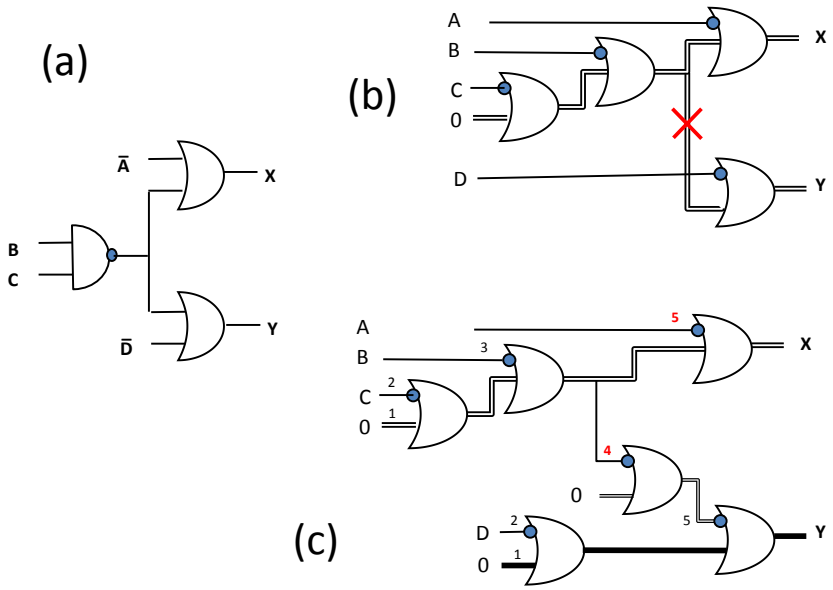


Figure 5. Fan-Out problem (a) NAND gate output going to two gates.
(b) Incorrect realization (c) fan-out problem fixed with additional gates

Figure 5(a) shows a simple circuit where the input of a NAND gate is provided to two OR gates. Figure 5(b) shows NAND gate replaced by two Imply gates, and two OR gates implemented using Imply gates. The circuit would be equivalent in CMOS logic. But in memristor logic, the working memristor (as shown by double lines) holds the output of NAND gate. In next pulse, the output is OR'ed with the negation of A, and the value of memristor is now changed. The original output of NAND gate is lost and cannot be provided to the second OR gate below. Figure 5(c) introduces additional gates and working memristors to fix the problem. Notice that the output from first NAND gate is now used at pulse 4 *before* the state of the memristor gets changed in pulse 5.

This unique characteristic of memristor circuits requires new Computer Aided Design (CAD) synthesis tools and a new notation to accurately represent the Memristor based Imply gate diagrams.

IMPLY SEQUENCE DIAGRAM

In this report, I will use the new "*Imply Sequence Diagram" (ISD)* notation that I invented for the application of analysing and synthesizing the memristor circuits. In this notation, horizontal lines represent physical memristors, while the ⊥– symbol represents a pulse applied to it. The top side of this symbol is the negated input. The left side is the non-negated input and the value of

memristor before the pulse. The right side is the value of the memristor after the pulse. A 0 on the left side indicates an additional pulse required to reset the input state of the memristor to 0.

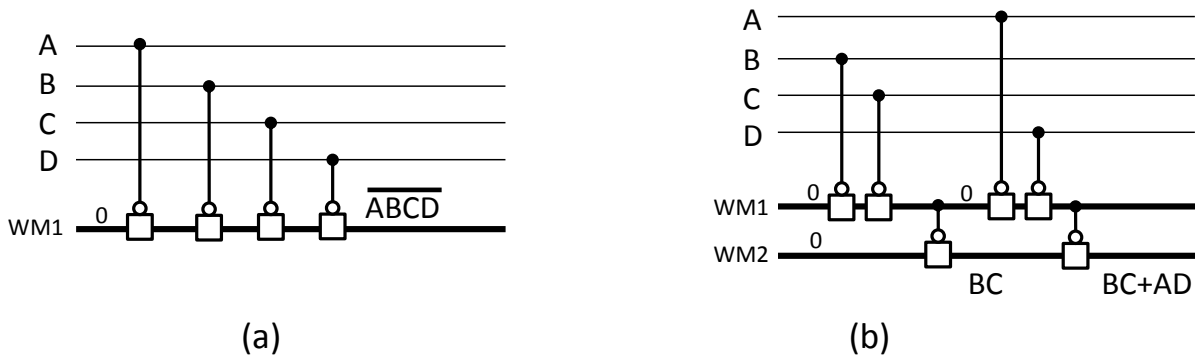


Figure 6. Imply Sequence Diagram (ISD Notation.
(a) synthesis using one working memristor. (b) a different example with two working memristors.

Figure 6 shows the ISD notation for two example circuits. The first one is a 4-input NAND gate using one working memristor. It can be seen that an n-input NAND gate can be easily realized by applying the appropriate input at subsequent pulses. A 4-input NAND gate can be realized with 5 pulses as shown in the diagram. The second example shows a (unate) Sum of Products BC+AD and requires 8 pulses.

This new notation helps draw the circuits that memorize all intermediate states and reuse virtual gates. It resembles the Feynman notation known from reversible and quantum circuits. In these two types of circuits and the memristor-based IMPLY circuits, the result of the physical gate is stored in the memristor so that the same (physical) resource (memristor) is reused repeatedly by many virtual IMPLY gates. In this notation, the horizontal axis corresponds to time (number of pulses) and the number of horizontal lines corresponds to the number of physical memristors. This notation is useful to verify the number of WMs that are used at any moment of the synthesis process. In addition to a "two-WM synthesis" from **[Lehtonen09]** and this paper, and a "multi-WM synthesis" from **[Burger13]** in which any number of WM is possible, it is possible to design a method in which a given number K of "*allowed WMs*" is assumed. In such case the ISD diagram helps keep track of the current number of WMs, which is the number of horizontal lines in the diagram that are not input lines. This allows for synthesis of memristor circuits with any assumed number k of WM ($k \geq 2$). This assumption is motivated by future technologies that would allow $k > 2$ and uses the algorithm from this paper (one for $k=2$) as a subroutine.

Realizing a circuit in this notation allows the output of an IMPLY gate (represented by a line in the ISD notation) to be given to many negated inputs of IMPLY but only to one non-negated input which is in the same line. Therefore, by using this notation the designer does not have to solve the fan-out problem. The ISD notation does not allow for direct connections between inputs and IMPLY gates. It clearly orders the designer to use another IMPLY as a negation to copy the variable A.

The ISD notation allows for a simple investigation of the trade-offs between the number of WMs and the number of pulses for various types of logic implemented with IMPLY gates. This is similar to the trade-offs between the number of ancilla qubits and two-qubit gates in quantum circuit synthesis. Similar to reversible notation, ISD allows for the derivation of binary matrices of circuits using matrix multiplication sequentially in inverse order for serial connections and Kronecker multiplication for parallel connections of subcircuits. The only required primitives are a 4*4 matrix of IMPLY gate and a 2*2 matrix of FALSE (zero) gate (these matrices, in contrast to reversible logic are not permutative, otherwise all the methods are exactly the same).

An important trait of memristors is that the output of the IMPLY gates is stored and is only changed if a pulse is applied. Because of this trait, it has been shown that any single output, *n*-input Boolean function can be represented with *n "input memristors"* and only two "*working memristors*" **[Lehtonen09, Lehtonen10].** The role of a WM is to store the intermediate data. In contrast to classical CMOS logic in which combinational and sequential primitives exist, every logic operator in memristor technology stores the data. There is no need to separate the combinational and sequential logic concepts; therefore I believe that some new notations and design methodologies for sequential circuits, state machines, iterative circuits and pipelined/systolic systems with memristors should be invented. In this case, they are all based on generalizations to ISD, but other possibilities should also be investigated in the future.

Imply Sequence Diagram notation in Figure 6(a) shows a circuit that only physically uses one working memristor. The squares represent the reuse of this memristor in time. The memristor is able to use the output of the previous cycle and use it as an input to the next cycle. The figure illustrates that a NAND of *n* inputs can be built with one WM and *n+1* pulses. We also see that the design process with IMPLY gates differs from the classical combinational synthesis and it is more similar to scheduling and allocation of operators (virtual IMPLY gates in this case) to physical resources (memristors in this case).

More information on memristors, IMPLY gates and generation of timing pulses can be found in **[Burger12, Borghetti10, Kvatinsky11, Poikonen12**].

LOGIC SYNTHESIS TO MINIMIZE THE DELAY IN MEMRISTOR NETWORK WITH TWO WORKING MEMRISTORS

Because one can build gates like OR, AND, NAND, EXOR and NOT from IMPLY gates, any known synthesis method can be adopted to design logic circuits with memristors. In particular, the synthesis methods such as: Sum-of-Products networks (SOP), Product-of-Sums (POS), TANT and NAND networks **[Gimpel67]**, linear, bi-decomposition **[Mishchenko01]**, Ashenhurst-Curtis decomposition**,** Reed-Muller, ESOP and negative gate circuit synthesis methods can be adopted and I have initially compared some of these methods on several types of single-output functions such as unate functions, parity functions and affine functions of many variables, cyclic functions, non-cyclic, balanced and self-dual functions. One can also try to develop exhaustive tree-search methods for the exact minimal number of pulses. However, when following [**Lehtonen09]** and **[Lehtonen10]** we want to synthesize circuits that have the exact minimum number of physical working memristors (which is related to the contemporary realization technologies as of 2012). Thus, the choices of synthesis approach become reduced, as we have to assume that only positive literals (variables) are used in the expressions **[Lehtonen09, Lehtonen10].** The papers of Lehtonen presented a canonical form with IMPLY gates which uses only positive literals in the expression. This algorithm follows the assumption of Lehtonen to use only two WM's, therefore this algorithm uses only the positive literals to realize the function. However, this canonical form is not minimal, as observed by Lehtonen himself. Moreover, this form is previously known from the research on synthesis with the minimum number of negative (complex) gates (**[Ibaraki71]** and several papers following it). In general, this form does not lead to the minimum number of virtual IMPLY gates, nor does it produce a circuit with the minimum number of pulses. Its main advantage is only giving a warranty of at most two WMs. Here a new method is presented that produces circuits that are never worse in the terms of pulse count than the circuits of Lehtonen. (Lehtonen does not present a final algorithm or program nor does he discuss experimental results, so I compare my results only with my interpretation of his method). The presented method attempts to minimize the number of pulses (IMPLY gates) assuming two working memristors (the same as

Lehtonen). My algorithm does not give a guarantee of a minimal number of pulses but the results are promising when compared to SOP and ESOP circuits mapped to IMPLY gates. Although the presented variant realizes only single-output functions, it is relatively easy to extend this method to multi-output circuits. This method can also be extended to more than two WMs (not discussed here, look to **[Burger13]** for similar methods).

The circuits synthesized by the algorithm MEMRMIN-2WM belong to the general class of circuits from **[Lehtonen10].** It results directly from the procedure below and the ISD notation that this procedure needs at most two WMs. The constructive design procedure is a simpler and more intuitive explanation of this fact than the formal proof from **[Lehtonen10].**

To make my explanation really simple, K-maps and standard logic schemata are used, but the software uses truth tables internally and a graphical form of ISD notation on the output. At first my method creates some groups of 1's (various types of the well-known *prime implicants*) which the method realizes with simple logic gates such as NAND. At any point, the largest group of 1's (a prime implicant) is taken, in which the literals are all *positive*. (A *positive literal* is a variable without a negation; a *negative literal* is a variable with negation). A *product implicant* is an implicant that is a product of literals. A product implicant is either a prime implicant or is included in a prime implicant. A product of positive literals from a product implicant (prime implicant) is called a *kernel*. A *positive prime implicant of function f* is a prime implicant of *function f* with all positive literals. Once the group is pinpointed, the gate is realized, and the 1's are replaced with dashes (don't cares) and may be used as any value in the next stages. This way, in addition to the original don't cares from the specification, the method adds more don't cares when realizing any prime of the function under synthesis. This property is especially useful when one synthesizes functions such as parity of many inputs. More and more don't cares appear in the maps, which reduces the number of IMPLY gates and consequently the number of pulses. If the algorithm, based on principles given below, decides that there are no more groups to be realized at this layer of the circuit (i.e. level of primes selection), the inverter has to be inserted into the circuit and the K-map must be negated. When realizing logic gates, I try to primarily use NAND gates and inverters because these can be easily expanded to IMPLY gates. OR gates are realized as sequences of IMPLY gates.

Here is the explanation of the key concepts to create a single layer of the circuit. By a layer I mean a sub-circuit that realizes a set of positive primes of *function f* or a set of positive primes of

one of the successively created remainder functions of *f*. An essential prime implicant (essential prime for short) is one that is the only possible prime implicant to cover a certain minterm (this minterm is called the essential vertex). The algorithm first realizes only the essential prime implicants that are also products of positive literals. These are called positive primes. If there are several positive primes in a layer of the circuit realization, only those that are essential are realized in this layer. If there are several positive primes for some minterm and none of them are essential, the largest prime (the one with the least number of variables) is realized. If all primes have the same size, one of them is randomly selected.

My algorithm realizes the circuit in layers starting from outputs of the circuit. For every layer, it finds the essential primes on positive literals and possible other primes on positive literals and replaces them with don't cares. When no possible groups for selection remain, the function is negated and the next layer is realized until the specification function of 0's is found.

Figure 7 shows the K-map of the function *f(A,B,C)* which needs to be realized using the presented logic synthesis algorithm. This *function f* is trivial, but is sufficient to explain the synthesis process.

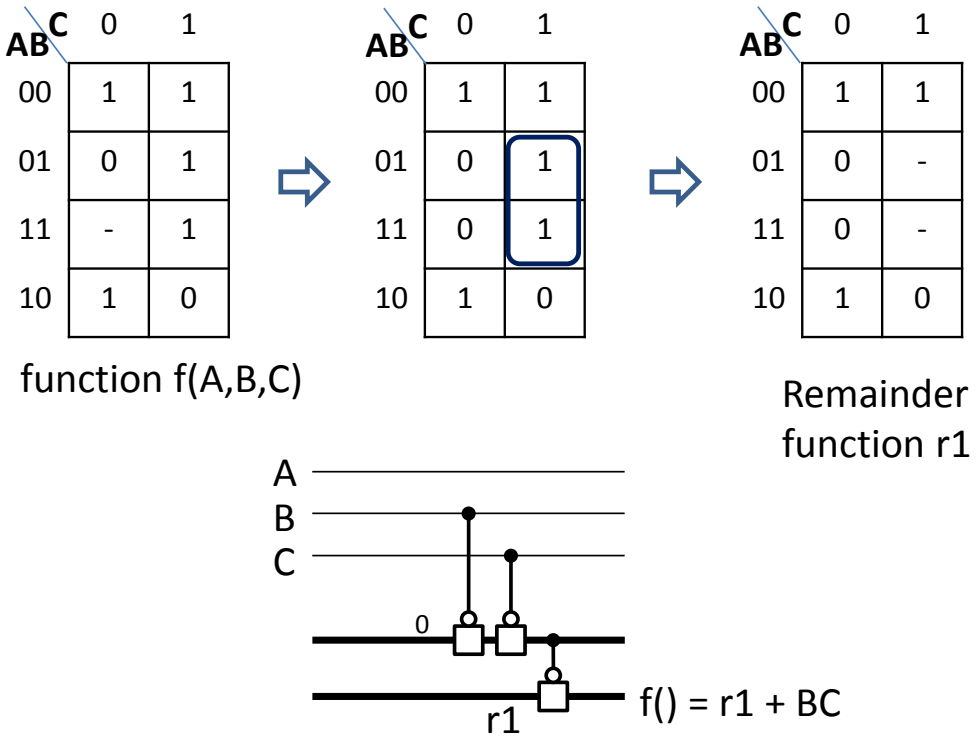


Figure 7: Function *f* shown in Karnaugh map, step 1 of realization.

The first step is to find the group of 1's in which the input literals are all positive. One such group is selected. In the example, this is BC. It is an essential positive prime, as there are no other positive primes covering minterm ABC (the essential vertex ABC).

BC is realized as a NAND gate and hooked to negative input of the IMPLY gate. So, at the end of this step -

$f(A,B,C) = r1 + BC,$

where r1 is the first remainder function. This is shown in the ISD notation in Figure 7.

Now the algorithm moves on to realize r1. Observe that there are no groups of 1's that correspond to positive variables. For example, the top row has two 1's, but this group is $\overline{A}\overline{B}$, so it does not qualify for selection. As there are no more positive primes, this completes the synthesis of the first layer. Now the entire K-map must be negated as shown in Figure 7 and r1 is replaced with a NOT gate and r2. In my software, negation of the K-map is simply reversing the roles of the temporary Onset and Offset tables.

At the end of this step, the function is realized by

$f(A,B,C) = \neg r2 + BC.$

The inverter is also realized with only an IMPLY gate. Figure 8 shows this stage.

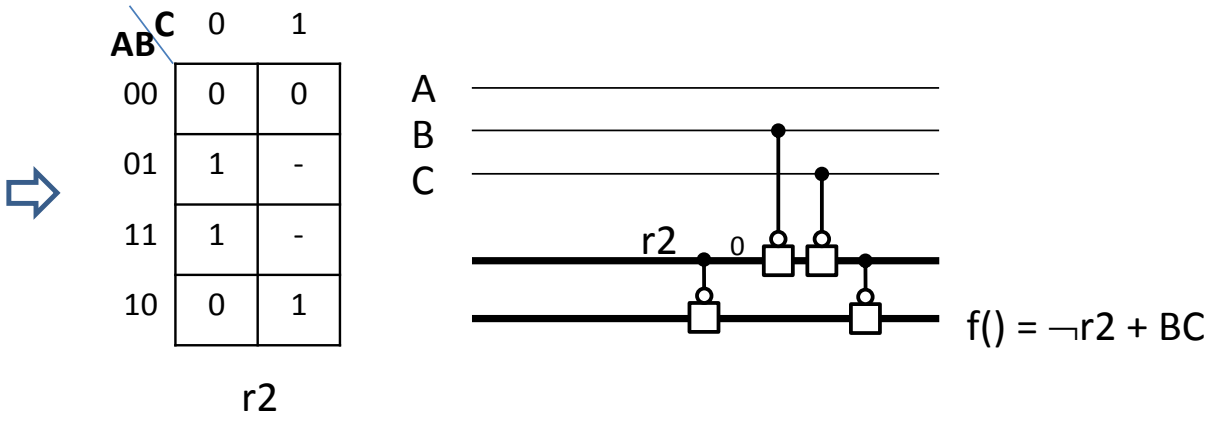


Figure 8: Inversion of r1 to r2

We can now find another group of 1's (and don't cares) which becomes the next positive prime implicant. This is shown in the Figure 9. Four middle squares are selected. These squares correspond to a positive essential prime B. Since this is a single variable, we put a NOT gate instead of NAND gate as shown. At the end of this step, the function realization is

$f(A,B,C) = \neg (r3 + B) + BC$

In this trivial example, the process of selecting the best essential prime implicant is not illustrated. This is done using the covering table and a Decision Function. It is important for realizing minimal circuit for larger examples. This is described separately in the next section.

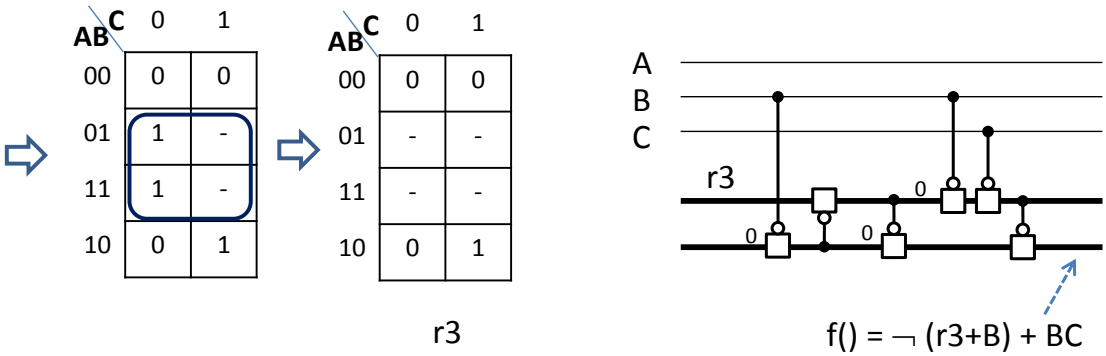


Figure 9: Realization of r2

There is still a '1' left in the K-map. To continue the process the remaining possible group (AC) is now selected, as shown in Figure 10. This positive prime is realized with an additional NAND gate. Group AC is again a positive essential prime so we do not need to use random choice. Since there is no more remainder function left, the algorithm puts a 0 at the remaining input of IMPLY gate.

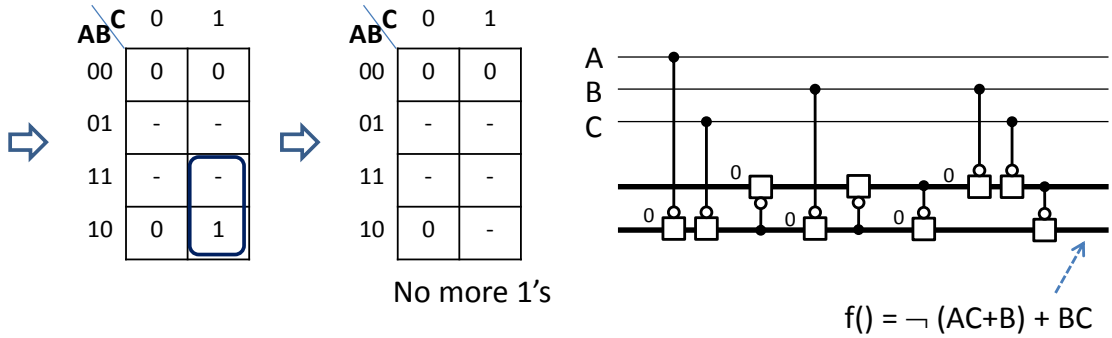


Figure 10: Final step of synthesis

The final function is now equivalent to

$f(A,B,C) = \neg (AC + B) + BC$

This is shown in the diagram in Figure 10. This function has two layers of positive primes, "BC" and "AC + B".

Figure 11 illustrates the pulse count in the circuit realized in Figure 10. The numbers at the top count the pulses; each cycle and each initialization have a pulse.

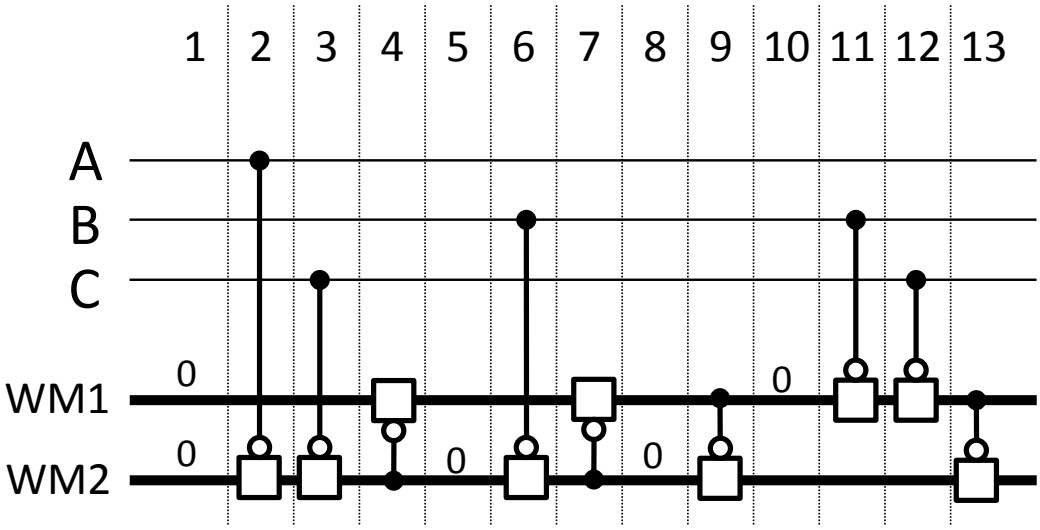


Figure 11: Counting the Pulses in the ISD notation

To illustrate the difference of my algorithm MEMRMIN-2WM and the method Lehtonen that takes all positive primes in a layer, the function g = ACD + AB$\overline{C}$ + $\overline{A}\overline{C}$D + $\overline{A}$BC will be realized. To realize the first layer the algorithm will select only positive prime ACD as it is an essential prime of essential vertex A$\overline{B}$CD. In contrast, Lehtonen's algorithm will create the first layer ACD + BD with a *redundant* positive prime BD.

MINIMIZATION ALGORITHM MEMRMIN-2WM FOR BINARY LOGIC

The software program implementing the algorithm reads the input from PLA files, K-map input files, or an equation in SOP format inputted as text string, and creates internal data structures. It is important to describe a few key data structures. All the minterms from the K-map are organized by the program into two sets. The OnSet array stores all minterms which have an output of 1. The OffSet array stores all minterms which have an output of 0. When a function needs to be negated, these two arrays can be simply switched. Once a set of minterms is realized, they go into don't care set, which is not stored. If a minterm is not in OnSet and not in OffSet, it is implicitly a 'don't care'. Positive prime implicants and essential prime implicants are stored in KernelSet array and used for each layer.

The following is the pseudo code of the core algorithm.

```
while OnSet is not Empty
    forall minterms in OnSet
        create kernel with only positive variables from minterm
        if kernel covers an element in OffSet
            discard kernel
        else
            add kernel to KernelSet (if it is not already there)
    end_forall
    sort KernelSet with reduced implicants on top (all prime implicants of this layer.
    find essential prime implicants in KernelSet and move them to top
    forall kernels in sorted KernelSet
        realize the kernel with an n-input NAND gate, and accumulate result with IMPLY gate
        remove all subsets of this implicant in KernelSet
        remove all minterms covered by this kernel from OnSet array
        if OnSet is empty
            break from while loop
    end_forall
    if OnSet is not empty
        negate the remainder function by switching OnSet and OffSet
```

add a NOT gate to the circuit
end_while
count pulses for all NAND, IMPLY and NOT gate
visually display results and pulse count

SOP METHOD USING MEMRISTOR IMPLY GATES

The software also implements the algorithm to use standard SOP expression and realize it using IMPLY gates. For a SOP expression with N input variables, the simplest algorithm uses N input memristors, N working memristors for negated input values, one working memristor for intermediate Product terms, and one working memristor for accumulating (summing) the product terms. Thus 2N+2 memristors are required. Figure 12 shows the implementation of SOP expression using this method. In another variant only three working memristors are needed, one used to create every negation of input variable, whenever this negation is necessary to realize a product.

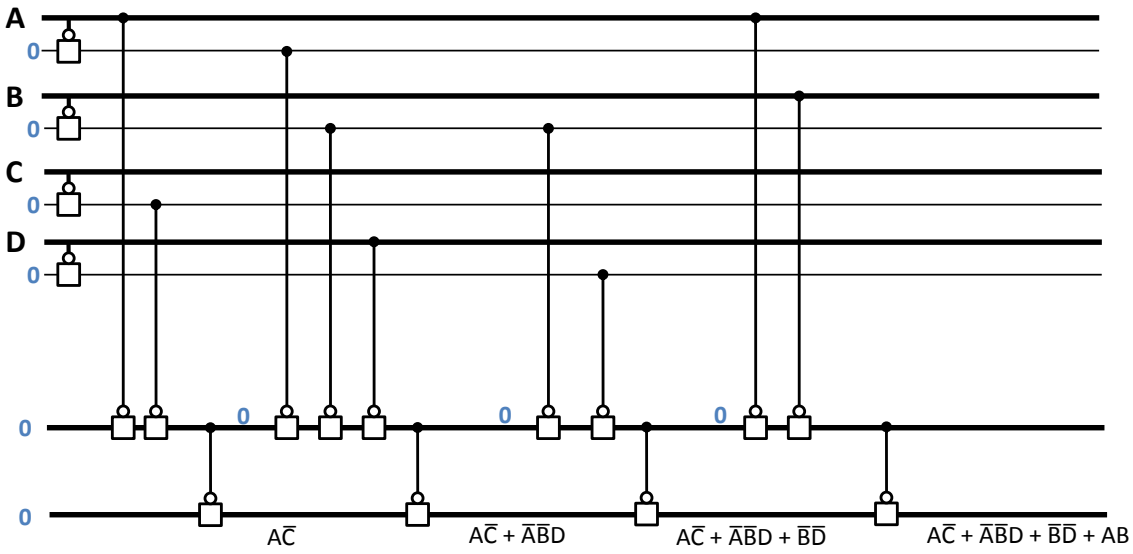


Figure 12: Realizing SOP expression ($A\overline{C} + \overline{A}\overline{B}D + \overline{B}\overline{D} + AB$) with N+2 = 4+2 working memristors

For the purpose of comparisons, the SOP expression was first minimized using Rondo tool and the minimized expression was then implemented using IMPLY gates to get the minimal necessary pulse count.

It is also possible to reduce the working memristors in the process of Figure 12. If the negated values are not stored, the N working memristors used for negation can be reduced to one. This

WM can be temporarily used to negate any variable that needs to be negated. This introduces one extra pulse for each occurrence of the negated variable in the SOP expression.

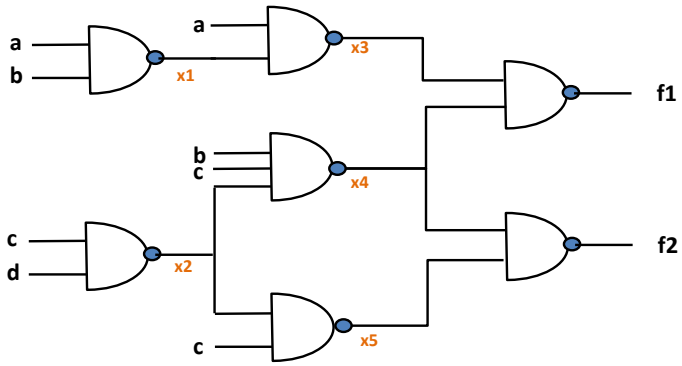
On the contrary, the number of working memristors in Figure 12 can also be increased. By adding two more working memristors, the SOP expression can be split into half, and two half expressions can be realized in parallel and combined in an extra step at the end. This reduces the pulse count dramatically. However, the implementation of this approach using standard CMOL-like CMOS/nano memristor cross-bars **[Lehtonen12]** may be difficult. Some special types of crossbars have to be invented that would allow for parallel pulses.

Another important point to note is that if the WM2 in Figure 12 can be observed for a value of '1', the evaluation can be stopped as soon as it turns '1'. Once the value is '1', it can never go to '0'. Therefore, the process could take a much less number of pulses to complete.

TANT NETWORKS USING MEMRISTOR IMPLY GATES

A TANT network is a universal three level network composed solely of NAND (AND-NOT) gates with only positive inputs. Figure 13 show a multi-output function using a TANT network. The two outputs realized are

- f1 = a(ab)'+bc(cd)' , and



- f2 = bc(cd)' + c(cd)'

Figure 13. An example of TANT (Three Level And-NOT network with True Inputs) Network

The TANT network can be converted to use Memristor Imply gates. Some care needs to be taken to address fan-out issue, as the intermediate signals labelled x2 and x4 are going to two different gates. The implementation of this circuit using the Memristor gates is shown in the ISD notation in Figure 14.

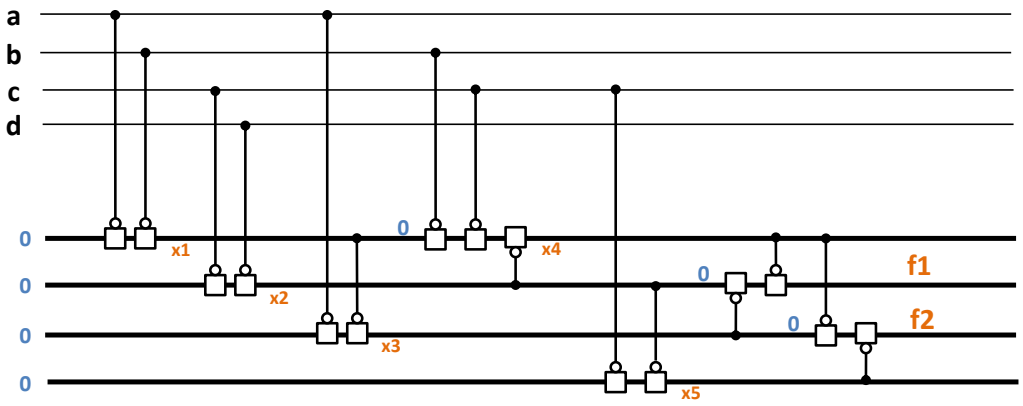
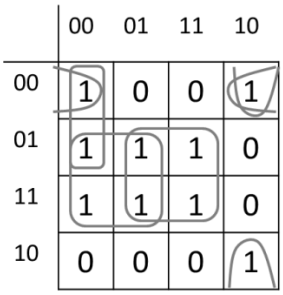


Figure 14. TANT network realized by Memristor Imply Gates

DECISION FUNCTION FOR SELECTION OF PRIME IMPLICANTS

MEMRMIN-2WM algorithm described above relies on the selection of best essential prime implicants to minimize the cost of the network. All other minimization algorithms such as SOP, POS, ESOP and TANT also require the selection of essential prime implicants. I address this step of the synthesis using covering table and Decision Function.

For MEMRMIN-2WM, POS and SOP, the problem is addressed using only the ‘covering table’. This results in the Decision Functions that only have positive literals. These are referred to as Unate functions. ESOP and TANT minimization methods requires the use of a closure table in addition to the covering table. The closure table provides additional constraints. This results in negated literals in the Decision Function. These functions are referred to as Binate functions. This process is described in more detail below.



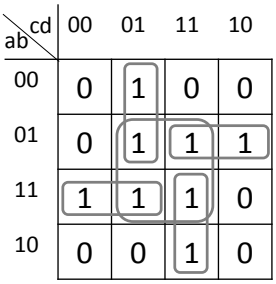
		0000	0100	0101	1100	1101	0010	1010	0111	1111
A	$\bar{a}\bar{b}d$	X					X			
B	$\bar{a}c\bar{d}$	X	X							
C	$b\bar{c}$		X	X	X	X				
D	bd				X	X			X	X
E	$\bar{b}c\bar{d}$						X	X		

Figure 15. A covering table for SOP Implicants

The Karnaugh map in Figure 15(a) is used here to illustrate the covering problem and decision function algorithm. The circled regions are the implicants. Certain implicants are necessary or 'prime' because one or more of the minterms are only covered by that implicant. However, some of the implicants are nonessential because all minterms are covered by other implicants. To realize functions like these, it is ideal to take the least number of implicants in order to ensure efficiency. The problem described is called the covering problem and there exist many methods of solving it.

From the K-map, a covering table is created as shown in Figure 15(b). All implicants are listed as rows of the table. All positive minterms are listed as columns of the table. Each implicant is labeled, for example the implicant which covers bd is named E. For each positive minterm, an X is placed in the rows where the implicant covers that minterm. After creating a covering table, an expression is written based on the table. Each columnis examined, and the options of the implicants that cover that minterm are collected. For example, the minterm 0000 is covered by A and B. Hence either A or B must be taken to satisfy the function. Since all minterms must be covered, a product is written of the individual options to create an expression. For the example in Figure 15, the expression is (A+B).(B+C).(C).(C+D).(C+D).(A+E).(E).(D).

Another example is presented in Figure 16.



(a)

		0001	0101	0110	0111	1100	1101	1011	1111
A	$\bar{a}\bar{c}\bar{d}$	X	X						
B	$\bar{a}bc$			X	X				
C	acd							X	X
D	$ab\bar{c}$					X	X		
E	bd		X		X		X		X

(b)

Figure 16. Another example of covering table for SOP Implicants

For this example in Figure 16, the expression derived from the covering table would be - (A)(A+E)(B)(B+E)(D)(D+E)(C)(C+E).

Now we apply Decision Function algorithm to select the labeled implicants (literals) from the above expression. This can be realized using various branching methods to find the optimal solutions. My program uses different Decision Function methods.

Method 1 is an exhaustive search which uses recursion to branch setting a literal to 1 to create a branch, and then creating all possible branches. Setting a literal to 1 implies that the corresponding implicant is picked. The expression is now reduced. For example, the expression (A+B)·(B+C) · (C) · (C+D) · (C+D) · (A+E) · (E) · (D) reduces to (B+C) · (C) · (C+D) · (C+D) · (E) · (D) if A is set to 1. The branching continues until the entire expression reduces to 1. The selected literals in the branch is a solution. This method finds all solutions and then eliminates solutions which are not optimal, leaving an array of optimal solutions. Solutions which are not optimal would include ones which are subsets of others. For example, if A and AB are two different soltuions, Method 1 would eliminate AB from the array of optimal solutions. However, if BC was another solutions, this solution would be kept because it is not a subset of A or AB. This method also works with negated variables.

Method 2 is a heuristic approach. The heuristic approach is my preferred method because of efficiency. The first step of this algorithm is to select all positive literals that are standalone. For example in expression (A+B).(B+C).(C).(C+D).(C+D).(A+E).(E).(D), three literals C, D and E are selected. Then the algorithm branches for literals which occur most often in the expression. The most occuring variable or variables are set to 1, and the same process is repeated on the

reduced expression. As opposed to Method 1, this method finds all optimal solutions automatically instead of finding all solutions and then narrowing down the list. This method does not work for satisfiability problem, because it may give a solution to a problem which in reality has no solution.

Using Method 2, the expression (A+B).(B+C).(C).(C+D).(C+D).(A+E).(E).(D) from first example is solved in two steps. In step 1, C, D and E are selected (as they are standalone), reducing the expression to (A+B). In second step A or B are selected. Thus, the solutions are CDEA or CDEB.

Similarly, the expression from second example, (A)(A+E)(B)(B+E)(D)(D+E)(C)(C+E) is solved in single step. Since A,B,C,D must be selected, the expression immediately reduces to 1, and ABCD is the solution. It is interesting to note that E (which is bd) is larger implicant that other four implicants, but it is not selected since it is already covered by other essential prime implicants (ABCD).

The decision function example above shows the Unate function that works for SOP and POS minimization. As I am looking at other different structures such as ESOP, multi level NAND networks and TANT networks, the minimization problem requires the use of covering-closure table, which leads to Binate functions. This is illstrated with a simple example in Figure 17.

This is a simple XOR gate implemented using a TANT function. The left side shows one way to select the implicants, where ab is subtracted from a to get one implicant, and ab is subtracted from b to get the second. On the right side, there is another way to select the implicants, where a is subtracted from b to get one implicant and b is subtracted from a to get the other implicant.

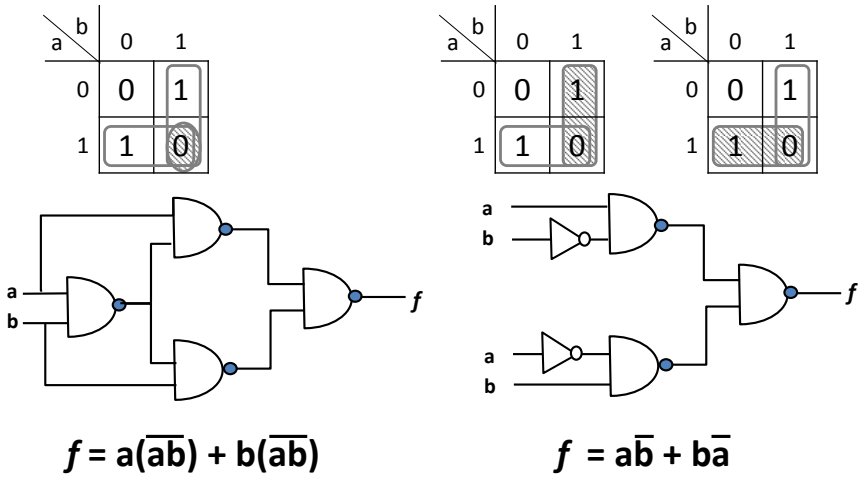


Figure 17. Another example of covering table for SOP

The set of implicants that can be used are $a\bar{b}$, $\bar{a}b$, !(ab), a and b. But if $\bar{b}$ is selected that implies that 'a' must be selected. This requires creation of the covering-closure table as shown in Figure 18.

		10	01	$\bar{b}$	$\bar{a}$
X	$a\bar{b}$	X		*	
Y	$b\bar{a}$		X		*
Z	ab			X	X
V	a				X
W	b			X	

Figure 18. Covering-Closure Table

The top left quadrant is the the covering table which is same as what I described earlier. All the rows are labeled with unique literals. Minterm 10 is covered by $a\bar{b}$ which is X label. Minterm 01 is covered by Y. The bottom right quadrant is the closure table. The rows are positive kernels of the implicants. To fill this part, each column of closure table is examined. In the example, to get the negated $\bar{b}$ implicant X must be selected, which is either ab' or equivalent to a(ab)'. So, b' can be derived from either b or ab. So in the closure table, either Z or W must be selected. Similarly to cover $\bar{a}$, Y implicant is selected, which is equivalent to ba' or b(ab)'. This requires either Z or V to be selected. This translates into the following decision function. X.Y.(X→Z+W).(Y→Z+V)

The imply operation (A→B) can also be written as $\bar{A}$+B. This the decision function can be re-written as X.Y.($\bar{X}$+Z+W).($\bar{Y}$+Z+V).

This is now a Binate function with negated literals. My decision function work well for such Binate functions as well. The algorithm will select X and Y as the implicants. This is a trivial case to illustrate how the decision functions are created in TANT minimization problem. A bigger function should be used to illustrate the benefit of the decision function solving algorithm. My continued research will use my decision function algorithm heavily for minimzation of different kind of multi-level logic structures.

EXPERIMENTAL RESULTS FOR BINARY BENCHMARKS

The presented algorithm was applied to several single-output benchmarks from ISCAS and MCNC PLA test sets. The PLA benchmark input format consists of terms, mainly those of which have an output of one. Each input term is some combination of ones, zeros, and don't cares. Each benchmark is analysed according to the algorithm and the output is displayed graphically as well as numerically. The pulse counts are recorded.

Table 1 shows the numerical results from the synthesis. The program is very time efficient, taking only a few milliseconds for the benchmark with 16 variables and 1547 terms.

Table 1 shows the pulse counts from the MEMRMIN-2WM algorithm in the column labelled Method1. For comparison purposes, I also used two other synthesis tools. Exorcism4.exe **[Mishchenko01a**] provides efficient ESOP minimization. This tool was used on the same PLA files. I then created another program to count the pulses required to realize the optimized ESOP circuit. Pulse counts used for NAND and NOT gates were the same as what was used in Method1. Each negated literal as an input had additional cost of 2 pulses (NOT gate). Each EXOR gate has a pulse count of 6. The pulse count from this is shown in the column labelled Method2. The second tool that was used was Rondo.exe from UC Berkeley. This tool provides efficient SOP minimization. All the PLA files were processed with this tool, and then another program was used to count pulses as was done for the ESOP circuits. The results from this experiment are shown in the column labelled Method3. A fourth experiment was conducted by taking the optimized output from Rondo.exe tool (SOP minimizer) and using it as the input to the MEMRMIN-2WM algorithm to find out if a combination of method would make my results even better. These results are recorded in the column labelled Method4.

TABLE I. PULSE COUNTS FROM 4 DIFFERENT METHODS

Method 1: Algorithm described in this paper (MEMRMIN-2WM)
Method2: ESOP Minimization (exorcism4.exe)
Method3: SOP Minimization (Rondo.exe)
Method4: MEMRMIN-2WM after SOP minimization

Benchmark File	Variables	Minterms	Results (Pulse Count)			
			Method1	Method2	Method3	Method4
exam1_d.pla	3	4	**32**	23	30	32
exam3_d.pla	4	6	**27**	30	35	27
rd53f1.pla	5	6	**30**	56	28	30
xor5_d.pla	5	16	**125**	41	190	125
rd53f2.pla	5	20	**56**	93	98	56
con1f1.pla	7	5	**35**	57	23	35
con2f2.pla	7	6	**23**	44	36	**17**
rd73f3.pla	7	35	**210**	329	208	210
rd73f1.pla	7	42	**142**	184	614	142
rd73f2.pla	7	64	**257**	61	1022	257
rd84f3.pla	8	1	**10**	8	8	10
newtag_d.pla	8	14	**12**	74	54	**9**
newill_d.pla	8	22	**42**	122	87	**29**
rd84f1.pla	8	120	**214**	245	1370	214
rd84f2.pla	8	128	**336**	73	2302	336
rd84f4.pla	8	162	**420**	674	628	420
max46_d.pla	9	47	**120**	877	881	120
9sym_d.pla	9	189	**420**	1112	1174	420
sao2f1.pla	10	10	**67**	223	212	67
sao2f2.pla	10	20	**49**	281	438	49
sao2f4.pla	10	85	**39**	245	313	**34**
sao2f3.pla	10	92	**41**	266	255	**20**
sym10_d.pla	10	837	**1260**	1912	2518	1260
t481_d.pla	16	1547	**121**	164	10464	**95**

The method described in this paper realized some of the functions with much less pulses. The highlighted rows show where the MEMRMIN-2WM method provided better results. This algorithm yielded better results for most functions. In general, this tool was advantageous for functions with a larger number of variables. This is important since most practical circuits will have larger number of variables.

Based on the results, on average my tool was better than all other tested tools in number of pulse counts. Another benefit of this tool is the working memristor cost, since it only uses two working memristors. The other tools were not optimized to reduce the number of working memristors. If that is done, the pulse counts for Methods 2 and 3 will perhaps further increase.

Obviously, Method4, which was a combination of SOP minimization followed by my algorithm, yielded some improvements. In all cases, the results from Method4 were either the same as in Method1, or better. This implies that Method4 can always be used as the best method. This

fact is obvious because, by doing SOP minimization first, only primes from a reduced cover are used and not all primes in every layer. Finally, a tool can be created that would select the best of all solutions from Methods 2, 3 and 4.

COST ANALYSIS FOR DIFFERENT LOGIC CIRCUITS

Let's look at different logic circuits and analyse the relative costs of these circuits if implemented using Memristor based Imply gates. The cost is determined by the number of memristors required and the pulse count required to implement the circuit. Let's start with a few illustrations.

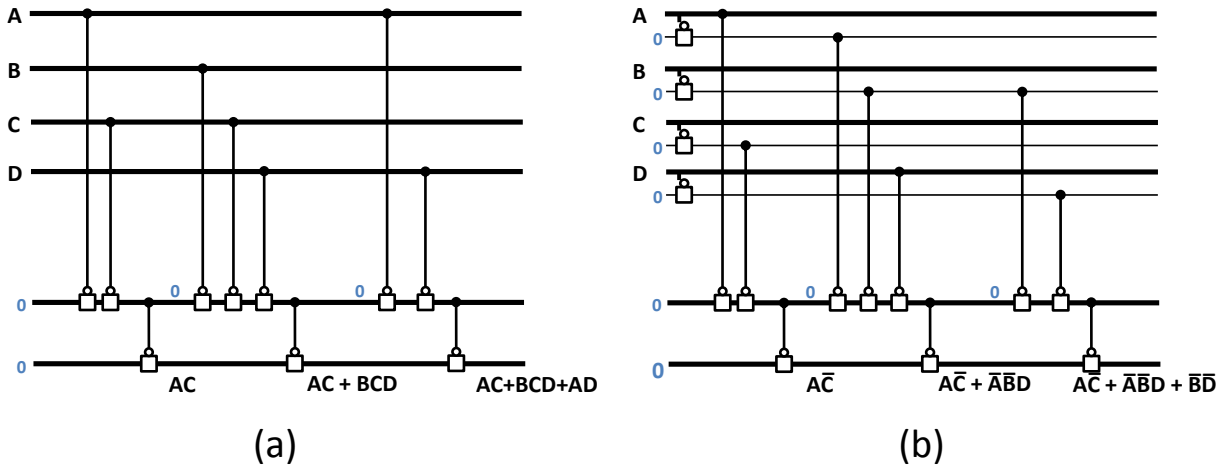


Figure 19. (a) Realization of Unate SOP with 2WMs, (b) Realization of general SOP with 6WMs

Figure 19 (a) shows a Unate SOP function. For this function, each product term is implemented by a NAND gate. Number of Imply gates required are equal to number of inputs to the NAND gate. So counting all positive literals (p) in the expression (7 for the example AC+BCD+AD) provides this count. In the ISD notation, these correspond to the vertical lines coming from the input memristors to the first working memristor. For each term, one extra pulse is required for initialization, and one pulse is required to accumulate the output of NAND gate into the second memristor. Thus a count of 2*terms is added to the count. This leads to the count of (p+2T) pulses for Unate SOP expression.

For a general SOP expression that can consist of negated variables, there are two ways to create the memristor circuit. If the inputs are negated once and stored in additional memristors, then no

additional pulses are needed, since the negation can be done on the same pulse when working memristors are being initialized. However L additional memristors are needed. This is shown in Figure 19(b). A second method is to use only one additional working memristor. This is used as a temporary memristor to negate a variable when needed. This adds one additional pulse for each negation.

Similar analysis is done for different kind of expressions. The results are shown in Table II. It gives the generalized cost for different functions using my methods. Values are calculated based on how many IMPLY gates are needed to realize each gate in the expression. In every case, the number of input variables equals the number of input memristors needed. The number of working memristors depends on what type of gates would typically be used to realize the expression.

The following notations are used in the Table II.

p = positive literals, e.g., for $A\bar{C}+\bar{A}\bar{B}D+\bar{B}\bar{D}$, this is 2
n = negative literals, e.g. for $A\bar{C}+\bar{A}\bar{B}D+\bar{B}\bar{D}$, this is 5
T = total terms, e.g. for $A\bar{C}+\bar{A}\bar{B}D+\bar{B}\bar{D}$, this is 3
L = unique input variables, e.g. for $A\bar{C}+\bar{A}\bar{B}D+\bar{B}\bar{D}$, this is 4 (A,B,C,D)
IM = input memristors
WM = working memristors

TABLE II. CALCULATING PULSE COUNTS FOR DIFFERENT FUNCTIONS

Type	Example Expression	Pulse Count	IM	WM
Unate SOP	AC+BCD+AD	p+2T	L	2
General SOP	$A\bar{C}+\bar{A}\bar{B}D+\bar{B}\bar{D}$	p+2n+2T	L	3
Unate POS	(A+B).(B+C+D).(E+A)	2p+T+2	L	3
General POS	$(A+B).(\bar{B}+C+\bar{D}).(\bar{E}+A)$	2p+n+T+2	L	3
Linear Parity Function	A⊕B⊕C	7T	L	3
Affine Parity Function	$A\oplus\bar{B}\oplus C$	7T	L	3
ESOP	$ABC\oplus\bar{A}\bar{B}\bar{C}\oplus ADE$	p+n+T+(T-1)*7	L	5
PPRM	ABC⊕DE⊕AD	p+T+(T-1)*7	L	5
FPRM	$ABC\oplus\bar{D}E\oplus\bar{B}C$	p+n+T+(T-1)*7	L	5
Using L extra Memristors to store one-time negated literals				
General SOP	$A\bar{C}+\bar{A}\bar{B}D+\bar{B}\bar{D}$	p+n+2T	L	2 + L
Unate POS	(A+B).(B+C+D).(E+A)	p+T+2	L	2 + L
General POS	$(A+B).(\bar{B}+C+\bar{D}).(\bar{E}+A)$	p+n+T+2	L	2 + L

The table shows the comparison of memristor circuits for various types of Boolean functions with example expressions. For instance, an arbitrary affine function needs only 3 WM and an arbitrary FPRM (Fix-Polarity Reed Muller) needs only 5WM. Pulse count is a function of unique input literals (L), total positive variables in the expression (p), total negative variables in the expression (n) and total terms in the expression (T).

These results may be useful to select appropriate synthesis methods and design new synthesis methods.

CONCLUSION

The presented method realized in program MEMRMIN-2WM is one of the first approaches to create a formal algorithm to minimize the number of pulses for a memristor-based logic circuit

that has the minimal number of WMs. This method does not assure a minimum solution, but it gives better results than the methods from **[Lehtonen09, Lehtonen10, Poikonen12]** based on manual computations (Lehtonen does not provide a data table with numerical results). My method also gives better results than **[Burger13]** on many functions. By analysing the solutions produced by this method, I found that further improvements of the presented algorithm are possible. In contrast to other methods, my approach allows for synthesizing circuits with don't cares. A higher percentage of don't cares corresponds to relatively better results.

I further expanded my research and synthesized other methods such as SOP and POS. I used Decision Functions to further minimize the cost. The Imply Sequence Diagram (ISD) notation that I created makes it simple to understand Memristor Imply gates, working memristors and pulse counts.

My current research as well as plans for future research include the following: (1) extension to multiple-output binary incompletely specified functions and assuming not necessarily only two WMs, (2) generalization of the method to multiple-valued (ternary) logic, (3) realization of POS (Product of Sums) circuits with minimum number of WM, (4) adaptation of the bi-decomposition algorithm **[Mishchenko01]** to concurrently minimize the number of WM and pulses **[Mishchenko01],** (5) adaptation of the method to fuzzy logic by using fuzzy maps instead of truth tables. I also found that the method from this paper can be easily extended to fuzzy logic and I plan to create real-life fuzzy benchmark functions (for instance from ML and robot control problems) and test the program on them. I plan also to investigate the trade-offs between the number of WM and the number of pulses for large benchmark functions.

WORKS CITED

[Borghetti10] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart and S. R.Williams, Memristive switches enable 'stateful' logic operations via material implication, *Nature*, vol. 464, pp. 873-876, 2010.

Authors show that nonlinear dynamical memory devices can also be used for logic operations and can be used to execute material implication (denoted by symbol →), which is a fundamental Boolean logic operation on two variables *P* and *Q* such that *P*→*Q* is

equivalent to $\bar{P} + Q$. Incorporated within an appropriate circuit, memristive switches can thus perform 'stateful' logic operations for which the same devices serve simultaneously as gates (logic) and latches (memory) that use resistance instead of voltage or charge as the physical state variable.

[Chua71] L.O. Chua, Memristor-missing circuit element, *IEEE Trans. Circuit Theory*, Vol. 18, pp. 507-519, (1971).

Dr. Chua was the first to describe the memristor, although only theoretically. In this paper he introduces memristor as the fourth basic circuit element and its peculiar characteristics which are different from that of resistors, inductors and capacitors.

[Davidson69] E. S. Davidson, An algorithm for NAND decomposition under network constraints, *IEEE Trans. Comput*., vol. C-18, pp. 1098-1109, Dec. 1969.

Author presents a branch-and-bound algorithm for the synthesis of multi-output, multilevel, cycle-free NAND networks to realize an arbitrary given set of partially or completely specified combinational switching functions.

[Gimpel67] J. F. Gimpel, The minimization of TANT networks, *IEEE Trans. Electron. Comput*., vol. EC-16, pp. 18-38, Feb. 1967.

A TANT network is a three-level network composed solely of NOT-AND gates (i.e., NAND gates) having only true (i.e. uncomplemented) inputs. The paper presents an algorithm for finding for any given Boolean function, a least-cost (i.e. fewest number of gates) TANT network.

[Kvatinsky13] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U.C. Weiser, Memristor-based Material Implication (IMPLY) Logic: design Principles and Methodologies, *IEEE Trans. Comput*. pp. 1-15, 2013.

This paper proposes a methodology for designing a memristor-based logic circuit is proposed. An IMPLY 8-bit full adder based on this design methodology is presented as a case study.

[Lehtonen09] E. Lehtonen and M. Laiho, Stateful implication logic with memristors. In *Nanoscale Architectures, NANOARCH'09. IEEE/ACM International Symposium on*, pp. 33-35, July 2009.

In this paper computation with memristors is studied in terms of how many memristors are needed to perform a given logic operation.

[Lehtonen10] E. Lehtonen, J.H. Poikonen, and M. Laiho, Two memristors suffice to compute all Boolean functions, *Electron. Lett.* 46, pp. 239–40.

This paper provides proof that all Boolean functions can be computed using two memristors.

[Lehtonen12] E. Lehtonen, Memristive Computing, *University of Turku,* pp.1-157, 2013.

This paper addresses the physical properties and the current-voltage behavior of a single memristor, memristor programming methods, and memristive computing in large scale applications.

[Mishchenko01] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions", *Proc. DAC '01*, pp. 103-108, 2001.

A new BDD-based method for decomposition of multi-output incompletely specified logic functions into netlists of two-input logic gates

[Mishchenko01a] A. Mishchenko and M. Perkowski, Fast Heuristic Minimization of Exclusive Sums-of-Products, *Proc. RM'2001 Workshop,* pp. 242-250, August 2001.

This paper presents an improved version of a heuristic ESOP minimization procedure.

[Poikonen12] 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 , vol.31, no.7, pp.1129-1134, July 2012.

This paper describes a procedure for representing any Boolean expression in a recursive form which can be realized using memristive devices, and demonstrates how the truth value of any Boolean expression can be determined using no more than two computing memristive devices.

[Strukov08] D.B. Strukov, G.S. Snider, D.R. Steward, and R.S.Williams, The missing memristor found, *Nature,* Vol. 453, pp.80-83, 2008.

Authors using a simple analytical example show that memristance arises naturally in nanoscale systems in which solid-state electronic and ionic transport are coupled under an external bias voltage.

WORKS CONSULTED

[Ashenhurst59] R.L. Ashenhurst. The decomposition of switching functions, *Computation Lab*, Harvard University, Vol. 29, pp.74-116, 1959.

[Burger13] J. Bürger, Ch. Teuscher, and M. Perkowski, Report. PSU. 2013.

[Burger12] J. Bürger, Disclosing the secrets of memristors and implication logic. Report. PSU. 2012.

[Dietmeyer69] D. L. Dietmeyer and Y. H. Su, Logic design automation of fan-in limited NAND networks, *IEEE Trans. Comput.*, vol. C-18, pp. 11-22, Jan. 1969.

[Ibaraki71] T. Ibaraki, S. Muroga, Synthesis of Networks with a Minimum Number of Negative Gates, *IEEE Tr. Comp*. Jan. 1971, Vol. 20 no. 1, pp. 49-58.

[Kvatinsky11] S. Kvatinsky, A. Kolodny, U.C. Weiser, and E.G. Friedman, Memristor-based IMPLY logic design procedure, *IEEE 29th Int. Conf. on Computer Design (ICCD)*, pp. 142–147, 2011.

[Laiho10] M. Laiho, and E. Lehtonen, Arithmetic Operations within Memristor-Based Analog Memory, 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA) 2010, Nanotechnology 23 (2012) 305205 (6pp).

[Linn12] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, Beyond von Neumann—logic operations in passive crossbar arrays alongside memory operations, *Nanotechnology*, Vol. 23, No. 30.

[Merrikh-Bayat11] F. Merrikh-Bayat, S. Bagheri Shouraki, Efficient neuro-fuzzy system and its Memristor Crossbar-based Hardware Implementation [cs.AI], 2011.

[Perkowski94] M. Perkowski, and M. Chrzanowska-Jeske, Multiple-Valued-Input TANT Networks, *Proc. 24th ISMVL*, pp. 334-341, 25-27 May 1994.

[Raghuvanshi13] A. Raghuvanshi and M. Perkowski, Synthesis of Incompletely Specified Logic Functions with Memristor-Realized Implication Gates, *presented at Reed-Muller conference*. April 2013.

[Shin11] S. Shin, K. Kim, and S. Kang, Reconfigurable Stateful NOR Gate for Large-Scale Logic-Array Integrations, *IEEE Trans. Comput*., vol.58, no.7, pp. 442-446, July 2011.