An Improved Mapping and Optimization Method for Implication-based Memristive Circuits Using And-Inverter Graph

Xiaoxiao Wang¹, Jiaxin Han¹, Yifang Yang², Yu Li¹

¹School of Computer Science, Xi’an Shiyou University, Xi’an 710065, China
xxwang@xsyu.edu.cn

²College of Science, Xi’an Shiyou University, Xi’an 710065, China
yangyifang@xsyu.edu.cn

Abstract. Any logic operation can be realized in implication-based memristive circuits characterized by low energy consumption and nanometre level size. The synthesis method transforms And-Inverter Graphs (AIGs) representation for logic functions into implication-based networks. It includes a mapping process which converts a cascade of nodes in AIGs to a series of implication gates and detects fan-out nodes simultaneously. An optimized copy process is employed to reduce the delay and area of memristive circuits in occurrence of a fan-out node. Experiments are carried out over a benchmark set including 33 functions with input variables from 3 to 41. Experimental results are compared with that from the original algorithm and another Majority-Inverter Graphs (MIG) based mapping method. It shows that the improved algorithm can obtain better performance in latency and area on most of the functions in the test set.

1. Introduction
Memristor remained as a theoretical concept reflecting the relationship of magnetic flux and electric charges[1] till the scientists in HP labs successfully built the first device in 2008[2]. This bipolar voltage-actuated switches can be used to execute material implication, which is a fundamental Boolean logic operation on two variables [3]. Memristor based logic circuit synthesis becomes a research hotspot due to its attractive characteristics in terms of scalability, high integration density, and non-volatility.

From the basic blocks point of view, memristive circuit synthesis uses imply gates [4-8] or other memristor-based stateful logic gate as basic blocks, such as NAND, NOR, NOT [9], multi-input implication [10] and Majority Gate [11].

Lehtonen and his team proved that it is only necessary to use \( n+m+2 \) memristors to implement an arbitrary Boolean function \( f: B^n \rightarrow B^m \). 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 [4]. By limiting the minimum number of memristor, the iterative sequential length was further reduced using the cost heuristic strategy or a directed acyclic graph [5-6]. Xiaoxiao Wang use evolutionary algorithm to tradeoff the delay and the number of memristors [7]. Marek Perkowski maps the And-Inverter Graph representation for a logic function to an imply gate
based circuit [8]. The similar idea is borrowed by [11] but the Majority-Inverter Graphs and Majority Gates are used instead. These two methods can be applied to multi-output and large scale functions.

From a technical implementation perspective, memristive logic circuits can be designed for pipeline or crossbar hardware structure. Above implication-based circuits are based on pipeline structure. Crossbar-based designs usually realize minterms simultaneously to reduce the latency but need more complex CMOS controller in order to compensate the driving force and support more execution states [12].

This paper aimed at proposing a mapping method for AIG to imply cascades conversion different from [8], including a new fan-out detecting approach and optimization steps when appearing fan-out nodes. We call it AIG-IMP for short. In [8], there is no practicable fan-out detecting method and systematic mapping and optimization process. It is demonstrated through experiments that AIG-IMP can obtain a significant reduction of delay and area compared with [8] and [11].

2. Background

2.1. Material Implication logic and IMPLY gate

Material Implication (IMP) is a fundamental Boolean logic operation on two variables $p$ and $q$ such that $p$IMPL$q$ or $p\rightarrow q$ is equivalent to $(\neg p) \lor q$ or $p' + q$.

IMPLY (memristor-based material implementation) gate consists of two identical memristors M1 and M2 and one regular resistor (R), as shown in Fig. 1, together with auxiliary voltage driving circuits with input of V1 and V2. The resistance of R is chosen such that $R_{on}<<R<<R_{off}$. The logic status of M1 and M2, marked as $p$ and $q$, is considered as logic 1 and logic 0, if its resistance is $R_{on}$ and $R_{off}$ respectively.

![Fig. 1 Memristor-based IMPLY gate and truth logic](image)

$V_{clear}$ and $V_{set}$ are positive and negative voltage beyond two threshold levels, which are used to set a memristor to high resistance and low resistance respectively [2]. $V_{cond}$ is a bias voltage required for memristor logic operation. When $V1=V_{cond}$ and $V2=V_{set}$ are applied to M1 and M2 at the same time respectively, in case of $p=0$, M1 is at its high resistance status of $R_{off}$, $VR$ is close to 0, which sets the resistance of M2 to $R_{on}$, or $q=1$, regardless its previous status. When $p=1$, M1 is at its low resistance status. Therefore, $V_{f}=V_{cond}$. The voltage level on M2 becomes $(V_{set}-V_{cond})$. For properly chosen voltage levels, $|V_{set}-V_{cond}|<|V_{cond}|$, the status of M2 remains no change. $p'/q'$ present the logic status for memristors M1/M2 after IMPLY operation [3].

To reset a memristor to logic 0, it requires to hold $VR$ to 0, while applying $V_{clear}$ to V1 or V2. Both memristors of an IMPLY gate can also be reset to logic 0 by one single pulse in a similar way.

2.2. And-Inverter Graph

An And-Inverter graph (AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network. An AIG consists of two-input nodes representing logical conjunction, terminal nodes labeled with variable names, and edges optionally containing markers indicating logical negation.

It has been a renewed interest in AIGs as a functional representation for a variety of tasks in synthesis and verification [13]. In addition to combinational logic, AIGs have also been applied to sequential logic.
3. AIG-IMP algorithm

3.1. AIG-IMP algorithm description

**Input**: a file of a Boolean function in Berkeley logic interchange format (BLIF)

**Output**: pulse, area and a circuit composed of a series of imply gates, represented as \( g_i(p, q) \), where \( i \) is the index of an imply gate, \( p \) and \( q \) are the index of two memristors which conduct imply operation.

- **step1**: reading BLIF file, got the cascades of gates, represented as \( G_i(nf, nt) \), \( nf \) and \( nt \) are updated node ID. That means if we want to accomplish the \( i \)th imply operation or gate, we should apply \( V_{\text{cond}} \) to the M2 of the \( nt \)th Gate, \( V_{\text{set}} \) to the M2 of \( nf \)th Gate;
- **step2**: for each \( G_i(nf, nt) \), detect the type of fan-out;
- **step3**: map each \( G_i(nf, nt) \) into \( g_j(p, q) \) of Node \( N_i \) according to the type of fan-out, optimization is done simultaneously. where \( p \) and \( q \) are the index of memristor.

3.2. Data pre-processing to obtain input files

The inputs of AIG-IMP algorithm are files in Berkeley logic interchange format (BLIF) of each benchmarks, which can be obtained by using the synthesis tool ABC. First, command *read* within ABC is used to read a file in PLA format, then the optimization commands *strash, refactor* are employed to detect and merge equal subfunctions to reduce the size. Then, *map* command is applied to convert the AIG graph to a specified gate technology. Here imply. genlib library file is defined as follow to map conjunction and negation operations to imply Gates.

\[
\text{GATE zero 0 O=CONST0;}
\]
\[
\text{GATE imply1 1 O=(!a + b); PIN a INV 0.07 999 0.64 4.1 0.40 2.5 PIN b INV 0.07 999 0.46 4.1 0.37 2.5}
\]
\[
\text{GATE inv 1 0=!a; PIN * INV 1 999 0.9 0.3 0.9 0.3}
\]

where imply1 and inv represents two different imply operation \( a \rightarrow b \) and \( a \rightarrow 0 \) respectively. Fig. 2 shows output blif file for function *adder*.

![BLIF file for function adder](image-a)

![AIG graph for adder](image-b)

Fig.2 BLIF file and AIG from ABC for function adder

3.3. Node ID conversion

The node ID in BLIF file is start with 0 according to the order of inputs, outputs, and gates. While the value of an output is the value of one node or imply gate, such as, Cout and S0 is correspondence to n11 and n16 respectively, the updated node ID in our mapping system is start with 1 according to the order of inputs and gates.

Node ID conversion algorithm is as follow:

**Input**: a file in BLIF format
Output: PI, the number of primary input; PO, the number of output; $G_i(nf, nt)$, $i=1$ to size, size is the number of nodes in the mapping system.

step 1 $Pi=0$, $Po=0, i=1$;
step 2 if it is a line with .input, for each input, store the name of the input, set the index to i, $nf=i$, $nt=i$, that is, $G(i, i)$, $i++; PI++$;
step 3 if it is a line with .output, for each output, store the name of the output, and $PO++$;
step 4 if it is a line with .gate, for each gate, do the following steps;
step 5 if the value of $a$ is the name of PI or PO, $nf$ is equal to the index of corresponding PI or PO, else $nf=x−PO+1, x$ is the subtracted index of original node;
step 6 if the value of $b$ is the name of PI, $nt$ is equal to the index of corresponding PI, else if it is empty, $nt=0$, else $nt=x−PO+1, x$ is the subtracted value from nx;;
step 7 if the value of o is the name of output, set the index of corresponding output to i;
step 8 set $G_i(nf, nt)$, if it is not the last gate, $i++; go to step 3$, else go to next step;
step 9 size=i;

For adder function in Fig 1, we can obtain the following results after node ID conversion: $PI=3$, $PO=2$, $size=15$, $G1(1,1)$ $G2(2,2)$ $G3(3,3)$ $G4(1,0)$ $G5(3,0)$ $G6(5,2)$ $G7(6,4)$ $G8(2,5)$ $G9(8,0)$ $G10(7,9)$ $G11(6,9)$ $G12(11,4)$ $G13(4,11)$ $G14(13,0)$ $G15(12,14)$.

3.4 Fan-out detecting

Imply gate $G_i(nf, nt)$ means we changed the value of the $nt$th node through imply operation, if we meet another $G_j(nt, *)$ or $G_j(*, nt)$ subsequently, that means we want to read or change the value of original $nt$ th node, that means the $nt$ th node is a fan-out node. For example, when $G8(2,5)$ is detected, we can figure out 2th node is fan-out, because $G5(5,2)$ changed the value of node 2. Similarly, 9th and 4th nodes are also fan-out. However the situation is not so simple, if we add $G16(2,13)$ $G17(14,2)$, the number of fan-out for 2th node is 1, because when we meet $G8(2,5)$ we copied 2th Node already. If add $G16(14,2)$ $G17(2,13)$ the total number of fan-out for 2th node is 2.

Input: $G_i(nf, nt)$, $i=PI+1$ to size, 0/1 matrix $fan[size][size+1]$ is initialized to 0

Output: the fan-out type of $G_i(nf, nt)$.

step 1 set type=0;
step 2 for $k=1$ to $i$, if $fan[k][nf]$ is equal to 1, clear it and set type to 1;
step 3 if $nf!=0$, for $k=1$ to $i$, if $fan[k][nt]$ is equal to 1, then if type=0, set type=2; if type is 1, set type to 3;
step 4 set $fan[nf, nt]=1$;
step 5 return type.

Step 2 is used to detect the fan-out state of $nf$th node. $fan[k][nf]=1$ means the state of $nf$th node is already changed, $nf$th node must be copied before $G_i(nf, nt)$ wants to read its’ state. If there is an imply operation after $G_i(nf, nt)$ try to read or write to $nf$th node, it is not fan-out anymore, so $fan[k][nf]$ is cleared. Step 3 aims to detect the fan-out state of $nt$th node. The only difference is that it is unnecessary to clear $fan[k][nt]$ if it is equal to 1, because $G_i(nf, nt)$ will change the value of $nt$th node again after it is copied, it is still fan-out.

There are 4 types of fan-out situation for $G_i(nf, nt)$. If return value is 0, that means neither $nf$th nor $nt$th nodes are fan-out. The return value 1 means only $nf$th node is fan-out; 2 means just the $nt$th node is fan-out, 3 means both $nf$th and $nt$th nodes are fan-out.

3.5 Mapping and optimization in occurrence of fan-out nodes

Input: The cascade of $G_i(nf, nt)$, $i=1$ to size;

Output: pulse, area, the cascade of $N_o$, $i=1$ to size, each $N_o$ may contain one or more $g_i(p, q)$;

step 1 for $i=1$ to PI, map $G_i(i, i)$ to $g_i(i, i)$ and insert to $N_i$, respectively; pulse=0, area=PI;
step 2 for each $G_i(nf, nt)$, $i=PI+1$ to size, do the step3 to step 7;
step 3 if fan-out type is equal to 0 and, find the last gate g_x(u,v) in N_{nf}, let p=v; if G_i is an imply gate, that means n\text{t} is not 0, get the last gate g_x(s,t) in N_{nt}, let q=t, else if G_i is an inverse gate, q=area+1;

step 4 if type is equal to 1, optimize the copy of N_{nf} and return the value to p, the assignment method for q is the same with that of step 3.

step 5 if type is equal to 2, the assignment method for p is the same with that of step 3, optimize the copy of of Nnt and return the value to q.

step 6 if type is equal to 3, optimize the copy of N_{nf} and N_{nt} return the value to p and q respectively.

step 7 insert g_x(p,q) into N_i.

step 8 compute and output pulse, area and the encoding of the circuit.

When a fan-out node is detected, we usually copy it’s value to a new memristor through two imply gates so 2 pulses and 2 memristors should be added to the circuit. For example, when the mapping process meets G_8(2,5) and detects Node N2 is fan-out, adding g_{1}(2,4) to N_8 directly is not allowed, see Fig3. (a). If we copy N2, namely primary input a_0, then two gates g_{2}(2,6)g_{3}(6,7) should be added to N_2 to accomplish the copy, see Fig.3 (b). This process can be optimized through find the insertion location which can be any place before the N6, because N6 changed the value of N2. Here we just insert G_8(2,5) before N_6, because from the insert location to the end of the circuit by far, there exist a node N6 which need to read the value from N5, so we copy N5, then insert N8. N5 is a special imply gate (inversion), so its’ copy also can be optimized, and only 1 pulse and 1memristor are needed, see Fig.3 (c).

Suppose to replace G_6(5, 2) with G_6(4,2) in the above example, when G_8(2,5) is mapped, it can be insert before N6 directly because there isn’t exist a node after the insert location which need to read information from N5, see Fig 4(c). Hence, no additional pulse or memristor is necessary to copy N2.

4. Experimental Setting and Test Results

4.1. Experimental Setting

Two experiments will be conducted to compare the performance of AIG_IMP with other IMP based synthesis algorithms in terms of pulse and area, namely, circuit delay and the number of memristors, additionally, the time for mapping and optimization in AIG_IMP is also given. First, the results from AIG_IMP is compared with those from [7], [8] and [11] on one output benchmark functions from IWLS93. Then, the comparison is done between AIG_IMP and MAJ_IMP [11] on multi-output functions from ISCAS89 and LGSynth91. These benchmarks spread in a wide range of input number from 3 to 41, output number and circuit size.
4.2. **AIG_IMP on one output benchmark functions**

The comparison is done between IMP mapping from MIG method MIG_IMP [11], original IMP mapping from AIG method AIG [8], evolutionary synthesis method IMP_MSEA [7] and AIG_IMP. The experimental results on 21 one output functions are listed in Table 1. AIG_IMP obtain less pulse and area than MIG_IMP on 18 functions. It has better performance in terms of pulse than original AIG mapping method on 18 functions. Because the results about area had not be published in [8], comparison only on pulse seems meaningless. But it at least shows that the optimization process proposed to reduce pulse in AIG_IMP is valid. Almost all solutions from IMP_MSEA and AIG_IMP separately presents non-domination relationship in terms of pulse and area. It is shown that AIG_IMP can reduce pulse remarkably on one output function at the cost of area increasing.

Table 1. The comparison in terms of pulse and area between IMP based synthesis algorithms on one output benchmarks

| #  | function | Input | MIG_IMP[11] | AIG[8] | IMP_MSEA [7] | AIG_IMP |
|----|----------|-------|-------------|--------|---------------|--------|
|    |          |       | pulse       | area   | pulse         | area   | pulse | area | time(s) |
| 1  | exam1_d  | 3     | 44          | 43     | 12            | 19     | 8     | 12   | 8       | 0.004  |
| 2  | exam3_d  | 4     | 55          | 50     | 12            | 18     | 9     | 17   | 12      | 0.006  |
| 3  | rd53f1   | 5     | 131         | 408    | 27            | 26     | 10    | 22   | 15      | 0.013  |
| 4  | rd53f2   | 5     | 77          | 77     | 57            | 62     | 10    | 44   | 26      | 0.019  |
| 5  | rd53f3   | 5     | 66          | 86     | 32            | 130    | 10    | 25   | 17      | 0.018  |
| 6  | 6mux     | 6     | -           | -      | 17            | 11     | 15    | 12   | 0.027   |
| 7  | con1f1   | 7     | 75          | 70     | 18            | 17     | 12    | 15   | 14      | 0.005  |
| 8  | con2f1   | 7     | 76          | 60     | 19            | 24     | 12    | 15   | 13      | 0.006  |
| 9  | rd73f1   | 7     | 121         | 291    | 238           | 402    | 12    | 183  | 92      | 0.175  |
| 10 | rd73f2   | 7     | 88          | 129    | 46            | -      | -     | 37   | 24      | 0.041  |
| 11 | rd73f3   | 7     | 107         | 193    | 104           | -      | -     | 36   | 24      | 0.053  |
| 12 | newtag_d | 8     | 96          | 90     | 21            | 35     | 13    | 15   | 13      | 0.005  |
| 13 | newill_d | 8     | 109         | 129    | 50            | 61     | 13    | 41   | 27      | 0.007  |
| 14 | rd84f1   | 8     | 153         | 430    | 351           | 1102   | 13    | 322  | 152     | 0.314  |
| 15 | max46_d  | 9     | 108         | 110    | 427           | 707    | 14    | 338  | 159     | 0.031  |
| 16 | 9sym_d   | 9     | 175         | 923    | 1418          | 1035   | 14    | 123  | 63      | 0.080  |
| 17 | sac2f1   | 10    | 108         | 110    | 102           | 94     | 15    | 77   | 46      | 0.010  |
| 18 | sac2f2   | 10    | 119         | 234    | 112           | 208    | 15    | 113  | 64      | 0.014  |
| 19 | sac2f3   | 10    | 143         | 325    | 380           | 130    | 15    | 53   | 34      | 0.008  |
| 20 | sac2f4   | 10    | 143         | 236    | 252           | 156    | 15    | 73   | 44      | 0.036  |
| 21 | t481_d   | 16    | 187         | 1285   | 1564          | -      | -     | 138  | 72      | 0.014  |

4.3. **AIG_IMP on multi-output benchmark functions**

Table 2 shows the delay and area from MIG_IMP and AIG_IMP on 12 multi-output functions. As expected, MIG_IMP can obtain less circuit delay on most functions because its’ mapping process consider one MIG level each time. It means all MAJ nodes in one level can realize simultaneously, which can reduce pulse largely but more memristors are required. AIG_IMP deals with each node in AIG serially and memristors are added when a fan-out or an inverse gate appears, so more delay and less area are needed compared with MIG_IMP whose delay and area are about ten times the MIG level and six times of the number of MAJs in the level with the most MAJs.
Table 2. The comparison in terms of pulse and area between MIG_IMP and AIG_IMP on multi-output benchmarks

| #  | function  | Input | output | MIG_IMP | AIG_IMP | pulse | area | pulse | area | time(s) |
|----|-----------|-------|--------|---------|---------|-------|------|-------|------|---------|
| 1  | 5xp1_90   | 7     | 10     | 99      | 101     | 207   | 199  | 207   | 199  | 0.013   |
| 2  | alu4_98   | 14    | 8      | 176     | 1087    | 2457  | 2160 | 2457  | 2160 | 1.020   |
| 3  | apex2     | 39    | 3      | 143     | 345     | 720   | 531  | 720   | 531  | 0.065   |
| 4  | clip      | 9     | 5      | 110     | 138     | 293   | 312  | 293   | 312  | 1.020   |
| 5  | cm150a    | 21    | 1      | 77      | 64      | 89    | 147  | 89    | 147  | 0.121   |
| 6  | cm163a    | 16    | 13     | 86      | 53      | 90    | 82   | 90    | 82   | 0.014   |
| 7  | cordic    | 23    | 2      | 76      | 94      | 154   | 102  | 154   | 102  | 0.016   |
| 8  | misex1    | 8     | 7      | 66      | 67      | 125   | 111  | 125   | 111  | 0.012   |
| 9  | misex2    | 14    | 14     | 165     | 131     | 226   | 2207 | 226   | 2207 | 0.029   |
| 10 | parity    | 16    | 1      | 132     | 55      | 90    | 216  | 90    | 216  | 0.012   |
| 11 | seq       | 41    | 35     | 153     | 1775    | 4126  | 3189 | 4126  | 3189 | 0.444   |
| 12 | x2        | 10    | 7      | 76      | 51      | 83    | 66   | 83    | 66   | 0.138   |

5. Conclusion

Based on the results and discussions presented above, the conclusions are obtained as below:

1. Generality: AIG_IMP is potential to map arbitrary logic circuits as long as they are based on Boolean logic.

2. Scalability: AIG_IMP is scalable to map logic circuits as large as possible.

3. Tradeoff between delay and area: In order to trade off the two indexes and simplify the technique realization of memristor based circuits from the control circuit view, the parallelization of nodes mapping deserves to be studied further.

Acknowledgments

This work was financially supported by the Shaanxi Provincial International Co-operation and Exchanges in Science and Technology Plan Project (2016kw-047), Natural Science Basic Research Plan in Shaanxi Province of China (2018JM6093), and Scientific Research Plan of Shaanxi Committee of Education (17JK0595, 17JK0610).

References

[1] Chua, L.O.: ‘Memristor – the missing circuit element’, IEEE Trans.Circuit Theory, 1971 (5):507–519

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

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

[4] E. Lehtonen, J.H. Poikonen, and M. Laiho, “Two memristors suffice to compute all Boolean functions”, Electronics Letters, 2010, 46 (3): 230-231.

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

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

[7] Wang X , Tan R , Perkowski M . Synthesis of memristive circuits based on stateful IMPLY gates
using an evolutionary algorithm with a correction function[C]/ IEEE/ACM International Symposium on Nanoscale Architectures. IEEE, 2016.

[8] J. Burger, C. Teuscher and M. Perkowski, “Design logic synthesis for memristors”, in Reed-Muller 2013, 2013

[9] Li Y, Zhong Y P, Deng Y F, et al. Nonvolatile “AND”, “OR”, “NOT” Boolean logic based on phase-change memory. Journal of Applied Physics, 2013, 114(23):114-117

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

[11] Shirinzadeh, Saeideh, et al. "Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs." Design, Automation & Test in Europe Conference & Exhibition 2016.

[12] Xie L, Du Nguyen H A, Taouil M, et al. A Mapping methodology of boolean logic circuits on memristor crossbar[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017:1-12.

[13] ABC- A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~lanmi/abc