Multi-Input MAGIC Synthesis and Verification for In-Memory Computing Design

Saeideh Nabipour[1] saeideh.nabipour@dfki.de
Kamalika Datta[1,2] kdatta@uni-bremen.de
Lennart Weingarten[2] len_wei@uni-bremen.de

Abhoy Kole[1] abhoy.kole@dfki.de
Rolf Drechsler[1,2] drechsler@uni-bremen.de

[1]*German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany*
[2]*Institute of Computer Science, University of Bremen, Germany*

***Abstract*—Recent advancements in memristor technology and *Resistive Random Access Memory* (RRAM) have made in-memory computing an alternative to tackle the limitations of traditional von Neumann architecture. Although significant progress has been achieved in the synthesis and mapping of Boolean functions within crossbar arrays using styles like IMPLY, MAGIC and Majority, verification processes have been relatively less explored. In this paper, a comprehensive method is presented for the synthesis, mapping and verification of multi-input NOR-based MAGIC in-memory design on RRAM crossbars. Our contributions are twofold: firstly, we extend crossbar micro-operations tailored for multi-input NOR logic enabling precise mapping. Secondly, a verification tool processes the modified representation and compares it against the golden reference design using *Boolean Satisfiability* (SAT) formula and *Satisfiability Modulo Theory* (SMT) solver. Experiments were conducted on the ISCAS'85 benchmark suite which shows the efficiency of multi-input MAGIC NOR compared to the existing 2-input NOR-based MAGIC design.**

***Index Terms*—Memristor, Synthesis, Verification, MAGIC Micro-operations, Multi-input**

I. INTRODUCTION

The growing demand for efficient computing has emphasized in-memory computing with memristors and *Resistive Random Access Memory* (RRAM), enabling data processing directly within memory to reduce latency and enhance energy efficiency [1]. Memristors are promising candidates for logic design due to their unique ability to simultaneously store and process data. Furthermore, they offer key advantages, including non-volatility, high switching speeds and low power consumption. Their small size and consistent construction make them suitable for fabrication in a crossbar array [2]. Memristor research spans neuromorphic computing [3], [4], testing [5], [6], device modeling [7], [8], and circuit design [9]–[11], with the latter playing a crucial role in mapping logic functions to memristor crossbars.

Previous research [11]–[14] has primarily focused on developing algorithms and methods for the synthesis and mapping of logic functions, commonly employing Majority or MAGIC design styles. These studies primarily address synthesis results such as the required number of cycles and crossbar size. However, a major challenge persists: the lack of attention to verifying the accuracy of the mapping process. A comprehensive synthesis and verification process has been introduced for Majority-based design in [5], [15] and [16], as well as for MAGIC-based design in [6], specifically focusing on 2-input NOR-based MAGIC design. However, no comparable framework has been proposed for a generalized multi-input NOR MAGIC design style. This research addresses this gap by extending the crossbar micro-operation representation for multi-input NOR MAGIC mapping. Additionally, we develop a synthesis, mapping and verification process using the Z3 SMT solver to validate the design against a golden reference. Experimental results using the ISCAS'85 benchmark show the effectiveness of the proposed approach. The major contributions of this paper are summarized below:

- We extend crossbar micro-operations for multi-input MAGIC-based mapping, which can efficiently represent the operations to be performed on the crossbar.
- An equivalence checking flow is developed, where SAT formulas generated from the original (golden) specifications and their corresponding multi-input micro-operations are verified using the Z3 solver.
- We validate the proposed method by running ISCAS´85 benchmark suits [17], generating netlists of multi-input NOR gates for performance analysis.

The rest of this paper is organized as follows. Section II provides the necessary background and related works. In Section III we present the multi-input mapping and verification methodology. Section IV discusses the experimental results, followed by concluding remarks in Section V.

II. BACKGROUND AND RELATED WORKS

A. MAGIC Design Style

Memristor-Aided loGIC (MAGIC) has been introduced in [9], a technique where logic states are represented by resistance values. The NOR operation in the MAGIC design style is performed in two steps: first, the output memristor is initialized to 1 (high-resistive state); second, a voltage V_0 is applied to the input memristors, with the output connected to the ground. If both inputs are 0, the output remains at 1;

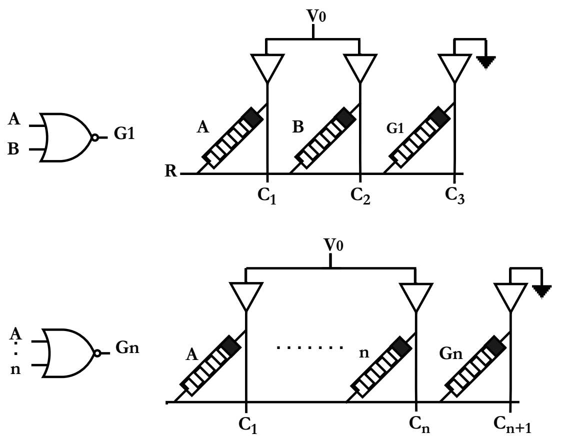


Fig. 1. MAGIC implementation of 2-inputs and n-inputs NOR gate.

if one or more inputs are 1, the output switches to 0. The MAGIC design style can only map NOR and NOT gates to the crossbar due to its structural constraints. However, since NOR is an universal gate, any logical operation can be decomposed into NOR gates. Fig. 1 shows the row-wise implementation of 2-input and n-input NOR gates, using one row with three columns for the 2-input gate and $n + 1$ columns for the n-input gate. Larger Boolean functions are synthesized using NOR and NOT gates, which are then mapped to the crossbar through various mapping techniques [11]–[14].

B. Related Works

Even though researchers have investigated the synthesis and mapping of Boolean functions to memristor crossbars [11]–[14], few studies have specifically addressed verifying the mapping process. Furthermore, most existing mapping and synthesis techniques do not automatically generate micro-operations. While synthesis and verification of majority-based logic design are the primary focus of studies such as [5], [15] and [16], their execution methods differ and are not directly applicable to MAGIC-based design. Specifically, [5] employs a Z3 solver for verification, while [16] uses *Decision Diagrams* (DD) to verify adder circuits. Research on MAGIC-based crossbar mapping [11], [13] has mainly focused on performance metrics such as crossbar size and cycle count, without addressing the representation of crossbar micro-operations. Notably, [11] examines the synthesis results for multi-input MAGIC NOR gates but does not provide a detailed analysis of crossbar-level micro-operations. More recently, the automated formal verification technique veriSIMPLER [6] has been introduced to verify the MAGIC-based synthesis and mapping method proposed in [11]. However, complete results for ISCAS'85 benchmarks were not provided. Furthermore, no comprehensive mapping and verification approach for multi-input NOR designs currently exists.

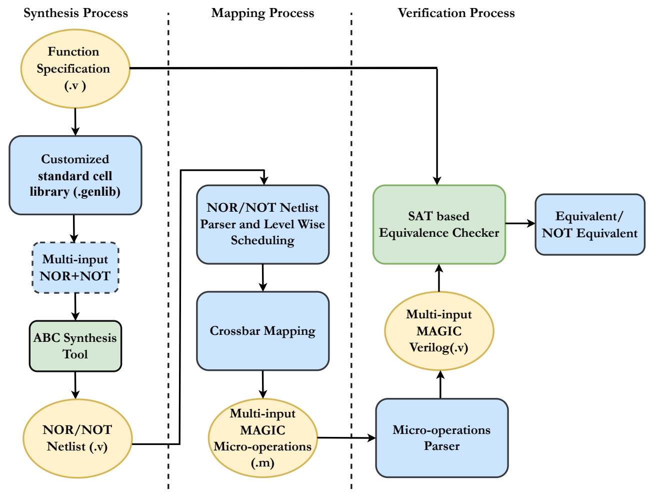

Synthesis Process
Mapping Process
Verification Process
Function Specification (.v)
Customized standard cell library (.genlib)
Multi-input NOR+NOT
ABC Synthesis Tool
NOR/NOT Netlist (.v)
NOR/NOT Netlist Parser and Level Wise Scheduling
Crossbar Mapping
Multi-input MAGIC Micro-operations (.m)
SAT based Equivalence Checker
Equivalent/ NOT Equivalent
Multi-input MAGIC Verilog(.v)
Micro-operations Parser

Fig. 2. The Multi-input MAGIC general flow of synthesis, mapping and verification.

III. Multi-input Synthesis, Mapping and Verification

This section outlines the proposed synthesis, mapping and verification method for multi-input MAGIC-based logic design in memristive crossbars. It first explains the mapping algorithm and micro-operation representation then details the verification process.

A. Overall Mapping Process

The general synthesis, mapping and verification approach is shown in Fig. 2, where the given function specification in Verilog form (.v) serves as the golden representation. The first step in the synthesis process is converting the Boolean function from Verilog into a netlist of NOT and multi-input NOR gates using the ABC synthesis tool [18]. For crossbar mapping, a level-wise scheduling algorithm, *As Late As Possible* (ALAP), partitions the gates into levels: $L_1, L_2, ..., L_L$. Next, a crossbar mapping tool maps the scheduled NOT/NOR netlist $(NL_{(NOT/NOR)})$ into the crossbar array, enabling parallel evaluation of independent gates within each level. Memristor micro-operations for gate evaluation are generated in a (.m) file. Notably, this study also addresses the mapping of buffers and constant inputs, which were overlooked in previous works [11], [13]. The mapping algorithm for Boolean functions in a memristor crossbar are detailed in Algorithm 1.

B. Multi-input Crossbar Representation

This section describes the crossbar micro-operation format generated by the synthesis and mapping tool for evaluating multi-input NOR MAGIC gates. The following cases specify the specifications for NOT and MAGIC NOR gate operations:

1) **NOT gate**: A NOT gate is equivalent to an n-input NOR gate with identical inputs. While mapping a NOT gate, two scenarios must be considered:
   a) NOT gate with primary input: the general format is:
   `<row_num> False <col1> </PI1> <col2>`

Algorithm 1: The Mapping Algorithm in Memristor Crossbar

```
1: Input: Function specification in Verilog form (.v)
2: Output: Micro-operations with extension (.m)
3: Begin
4: $NL_{(NOT/NOR)} = ABC\_Synthesis(F)$
5: $L_1, L_2, \dots, L_L = ALAP(NL_{(NOT/NOR)})$
6: for each Level do
7:     for each gate $g_i$ in Level do
8:         if ($g_i$ is NOT || NOR) then
9:             if ($g_i$- input $\in$ Primary Inputs ($PI$)) then
10:                MapParallel(row-wise) $PI$ to crossbar
11:            else
12:                MapParallel(row-wise) address of intermediate gate to crossbar
13:            end if
14:        end if
15:    end for
16: end for
17: End
```

`</PI1> ... <output_coln> True`

b) NOT gate with intermediate gate as input: the general format is:
`<row_num> False <col1> <`R_n×C_m`> <col2> <`R_n×C_m`> ... <output_coln> True`

2) **n-input NOR gate**: While mapping the NOR gate, three scenarios need to be considered:

a) NOR gate with primary inputs: the general format is:
`<row_num> False <col1> </PI1> <col2> </PI2> ... <output_coln> True`

b) NOR gate with intermediate gates as inputs: the general format is:
`<row_num> False <col1> <`R_n×C_m`> <col2> <`R_n×C_m`> ... <output_coln> True`

c) NOR gate with primary inputs and intermediate gates as inputs: the general format is:
`<row_num> False <col1> <`R_n×C_m`> <col2> </PI1> ... <output_coln> True`

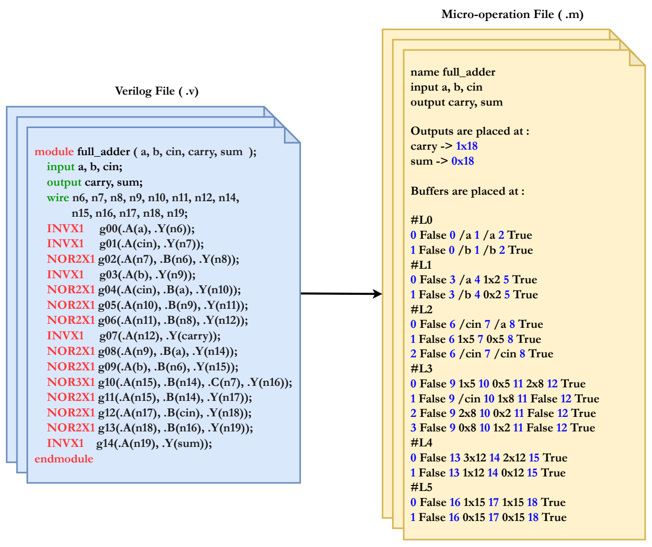
The `<row_num>` specifies a row number in the crossbar, with the value `False/True` applied to that row. `<col1>`, `<col2>`, etc., refer to specific column numbers. `</PI1>` and `</PI2>` represent primary input variables, as defined in the input Verilog file while R_n×C_m denote the locations of input gates in the crossbar i.e. n-th row and m-th column. If there are no dependencies among gate inputs, gates at the same level can be executed simultaneously after initialization. For multi-input NOR gates, it is important to consider the maximum number of inputs across all gates to ensure that they can be executed simultaneously without input dependency issues. If a gate has fewer inputs than the maximum input, a filler value of `False` is used for the remaining inputs. This ensures that all gates align with the maximum input requirement, enabling simultaneous processing. The output column for each gate is set to `True`, as the output memristors must be initialized to 1 for MAGIC evaluation.

An example is presented for full adder in Fig. 3 which represent the Verilog representation and NOR3 micro-operations. The topologically sorted netlist consists of six levels, with comments indicated by lines starting with #. The full adder requires no buffers, it has three inputs (a, b, cin), and two outputs (carry at 1 × 18 and sum at 0 × 18). Each gate requires $n+1$ crossbar columns—n for inputs and one for the output—where n is the maximum input count at each level. The total number of columns is determined by the maximum input count per level added with the output, while the highest number of gates at any level defines the required number of rows. The following six lines outline the MAGIC micro-operations for a full adder using NOR3 netlist on the crossbar:

1) At level 0 two primary inputs are negated. For this purpose two NOT gates are placed in row 0 and 1, sharing columns for parallel execution. The first gate uses `/a` as input (column 0 and 1) and sets column 2 to `True` for the output memristor, while the second NOT gate uses `/b` with the same column settings. The gates are then evaluated.
2) At level 1, two NOR2 gates use the primary inputs and outputs from the previous level. The first gate takes `/a` and memristor at 1×2 ($\neg/b$) as inputs, while the second takes `/b` and memristor at 0×2 ($\neg/a$) as inputs. Input values are read and copied to these memristors. Columns 3, 4, and 5 map the gates, and finally evaluation takes place.
3) Level 2 contains two NOR2 gates and one NOT gate. The first and third gates use primary inputs, while the second gate reads inputs from 1×5 and 0×5 and copies them to 1×6 and 1×7. The output memristors are initialized to `True`, and gate evaluation follows.
4) Level 3 includes one NOR3 gate and three NOR2 gates. The second gate uses primary input `/cin` and the memristor at 1×8 as inputs, while the other gates use inputs from previous levels. Since the maximum number of input is three, a filler value of `False` is added in column 11 for the second, third, and fourth gates. Output memristors are initialized to `True`, and the gates are evaluated accordingly.
5) Level 4 consists of two NOR2 gates. The first gate takes inputs from 3×12 and 2×12, then copies them to location 0×13 and 0×14. The second gate reads inputs from 1×12 and 0×12, and copies them to 1×13 and 1×14. Finally it is evaluated.
6) Level 5 consists of two NOT gates. The first gate takes its input from 1×15, and the second gate takes its input from 0×15. These inputs are then copied to locations 0×16 and 0×17 for the first gate, and to 1×16 and 1×17 for the second gate. After initialization, the gates are evaluated.

C. Verification Process

The first step in the verification process is to convert the original input specification and crossbar micro-operation into a format suitable for equivalence checking using a SAT solver. Here we are verifying whether the generated micro-operation


Verilog File (.v)
Micro-operation File (.m)

```
module full_adder ( a, b, cin, carry, sum );
  input a, b, cin;
  output carry, sum;
  wire n6, n7, n8, n9, n10, n11, n12, n14,
       n15, n16, n17, n18, n19;
  INVX1   g00(.A(a), .Y(n6));
  INVX1   g01(.A(cin), .Y(n7));
  NOR2X1 g02(.A(n7), .B(n6), .Y(n8));
  INVX1   g03(.A(b), .Y(n9));
  NOR2X1 g04(.A(cin), .B(a), .Y(n10));
  NOR2X1 g05(.A(n10), .B(n9), .Y(n11));
  NOR2X1 g06(.A(n11), .B(n8), .Y(n12));
  INVX1   g07(.A(n12), .Y(carry));
  NOR2X1 g08(.A(n9), .B(a), .Y(n14));
  NOR2X1 g09(.A(b), .B(n6), .Y(n15));
  NOR3X1 g10(.A(n15), .B(n14), .C(n7), .Y(n16));
  NOR2X1 g11(.A(n15), .B(n14), .Y(n17));
  NOR2X1 g12(.A(n17), .B(cin), .Y(n18));
  NOR2X1 g13(.A(n18), .B(n16), .Y(n19));
  INVX1  g14(.A(n19), .Y(sum));
endmodule
```

```
name full_adder
input a, b, cin
output carry, sum

Outputs are placed at :
carry -> 1x18
sum -> 0x18

Buffers are placed at :

#L0
0 False 0 /a 1 /a 2 True
1 False 0 /b 1 /b 2 True
#L1
0 False 3 /a 4 1x2 5 True
1 False 3 /b 4 0x2 5 True
#L2
0 False 6 /cin 7 /a 8 True
1 False 6 1x5 7 0x5 8 True
2 False 6 /cin 7 /cin 8 True
#L3
0 False 9 1x5 10 0x5 11 2x8 12 True
1 False 9 /cin 10 1x8 11 False 12 True
2 False 9 2x8 10 0x2 11 False 12 True
3 False 9 0x8 10 1x2 11 False 12 True
#L4
0 False 13 3x12 14 2x12 15 True
1 False 13 1x12 14 0x12 15 True
#L5
0 False 16 1x15 17 1x15 18 True
1 False 16 0x15 17 0x15 18 True
```

Fig. 3. Verilog and Micro-Operation File Format for Full Adder with Multi-inputs NOR gate.

actually realizes the specified function. The verification process is depicted in Fig. 2. First, a parser reads the micro-operations and converts them into Verilog specifications (*.v*). Using the ABC tool [18], we then create And-Inverter Graphs (AIGs) for both the original input in Verilog format and the Verilog file generated by the parser. Next, the AIGER tool [19] is employed to convert the AIGs into CNF clauses in DIMAC format. The AIGs from both files are first processed by AIGER, and then the `aigtoaig` command is used to generate (*.aag*) files for CNF clause creation. We then apply the `aigtomiter` command to the (*.aag*) files to generate a miter circuit. The miter circuit is represented in DIMAC format as CNF clauses. Once the DIMAC file is fed into the Z3 solver [20], the solver returns either SAT or UNSAT. If the mapping is equivalent to the original Verilog specification, the result will be UNSAT; if not, it will return SAT, indicating a mismatch between the mapping and the original specification. The verification process is further explained with an example.

Example 1. Consider a simple logic circuit shown in Fig. 4 ❶. Fig. 4 ❷ displays the reference Verilog design and the MAGIC-generated Verilog file. Using the ABC tool, both files are converted into AIGs in binary format (*.aig*), as shown in Fig. 4 ❸. The first line of the AIG file contains a header with the total number of nodes, inputs, latches, outputs, and AND gates. The following lines define the AND gates, with identifiers like i_0, i_1, followed by the label names. These binary AIGs are then converted to ASCII format (*.aag*), as shown in Fig. 4 ❹. A miter circuit is generated, which combines both AIGs in ASCII format (*.aag*) (Fig. 4 ❺), which is then used to generate the CNF format (*.dimacs*) for the Z3 solver, as shown in Fig. 4 ❻. The CNF file contains a header specifying the formula type, the number of variables, and the number of clauses, followed by clause representation as ORs of literals. Finally, Z3 confirms SAT/UNSAT in Fig. 4 ❼, indicating whether the two circuits are equivalent or not.

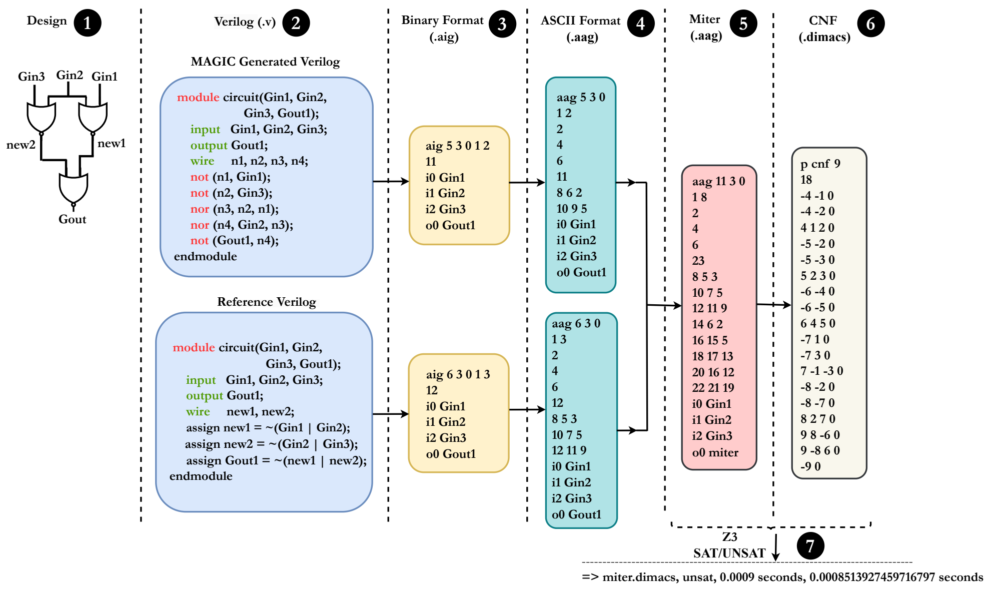
IV. Experimental Results

This section summarizes the experimental results. A set of functions from the ISCAS'85 benchmark suites [17] has been used. The mapping and parser tools which convert crossbar micro-operations into Verilog, are implemented in C++. Other tools like the ABC tool [18], the Python interface of the Z3 solver [20], and AIGER tool [19] have been used for synthesis and verification process. All the experiments have been run on a Thinkpad P14s with Intel i7-4750U CPU having 1.70 GHz clock and 40 GB RAM.

A. Mapping Results

The synthesis and mapping results using the ISCAS'85 benchmark suite [17] are shown in Table I. The first three columns list each circuit's name, the number of Primary Inputs (`PI`) and the number of Primary Outputs (`PO`). The next twelve columns show the synthesis results for NOR2, NOR3 and NOR4 netlists, including the crossbar size (`CBS`), total latency (L_T) (calculated as the sum of write and evaluation latencies), accurate total latency (L_{TA}) (calculated as the sum of read, write and evaluation latencies) and micro-operation generation time (S_t), reported in seconds. The following two columns list the crossbar size (CBS_S) and the total latency (L_{TS}) (calculated as the sum of initialization/writing '1' to the output memristors and evaluation latencies), as reported in SIMPLER [11] for NOR2 netlists. Finally, the last seven columns compare the crossbar size and latency of the proposed NOR2 mapping method with those of NOR3, NOR4, and SIMPLER [11], calculated as $NOR2_{\text{results-PM}}$ divided by either $NOR3_{\text{results-PM}}$ or $NOR4_{\text{results-PM}}$ and $NOR2_{\text{results-S}}$ divided by $NOR2_{\text{results-PM}}$, where results-PM refers to the proposed method and results-S denotes the SIMPLER [11] method. Moreover, the '-' indicates unavailable results for c17 benchmark in SIMPLER [11].

Table I shows that, among the eleven circuits, seven achieve better crossbar size, and ten achieve better latency with both NOR3 and NOR4 netlists compared to NOR2 netlists. Specifically, NOR4 netlists reduce the accurate total latency (L_{TA}) by 4.78% and 31.43% compared to NOR3 and NOR2, making NOR4 the best choice for latency-sensitive applications. This improvement is attributed to NOR4 requiring fewer gates, which reduces logic levels and consequently, the read, write, and evaluation times. However, NOR3 netlists are more compact, requiring 5.94% less crossbar area than NOR4, as the higher fan-in of NOR4 gates necessitates more memristor cells, resulting in larger crossbar sizes. Moreover, SIMPLER [11] is more area-efficient than our method because it maps operations to a single row of the memristor crossbar to achieve the required functionality, while also supporting cell reuse. However, our method achieves a 6.4× lower average latency when read latency is excluded from the calculation. To ensure a fair comparison with SIMPLER [11], which excludes read latency in its total latency evaluation, we analyze two scenarios: (i) total latency excluding read latency and (ii) accurate total latency including read latency. While our method achieves better latency reduction in the first scenario, it incurs a higher total latency when read latency is included, providing a more precise evaluation of overall latency compared to



```
module circuit(Gin1, Gin2,
               Gin3, Gout1);
   input   Gin1, Gin2, Gin3;
   output Gout1;
   wire     n1, n2, n3, n4;
   not (n1, Gin1);
   not (n2, Gin3);
   nor (n3, n2, n1);
   nor (n4, Gin2, n3);
   not (Gout1, n4);
endmodule
```

```
module circuit(Gin1, Gin2,
                  Gin3, Gout1);
   input   Gin1, Gin2, Gin3;
   output Gout1;
   wire     new1, new2;
   assign new1 = ~(Gin1 | Gin2);
   assign new2 = ~(Gin2 | Gin3);
   assign Gout1 = ~(new1 | new2);
endmodule
```

```
aig 5 3 0 1 2
11
i0 Gin1
i1 Gin2
i2 Gin3
o0 Gout1
```

```
aig 6 3 0 1 3
12
i0 Gin1
i1 Gin2
i2 Gin3
o0 Gout1
```

```
aag 5 3 0
1 2
2
4
6
11
8 6 2
10 9 5
i0 Gin1
i1 Gin2
i2 Gin3
o0 Gout1
```

```
aag 6 3 0
1 3
2
4
6
12
8 5 3
10 7 5
12 11 9
i0 Gin1
i1 Gin2
i2 Gin3
o0 Gout1
```

```
aag 11 3 0
1 8
2
4
6
23
8 5 3
10 7 5
12 11 9
14 6 2
16 15 5
18 17 13
20 16 12
22 21 19
i0 Gin1
i1 Gin2
i2 Gin3
o0 miter
```

```
p cnf 9
18
-4 -1 0
-4 -2 0
4 1 2 0
-5 -2 0
-5 -3 0
5 2 3 0
-6 -4 0
-6 -5 0
6 4 5 0
-7 1 0
-7 3 0
7 -1 -3 0
-8 -2 0
-8 -7 0
8 2 7 0
9 8 -6 0
9 -8 6 0
-9 0
```

```
--------------------------------------------------------------------------------
=> miter.dimacs, unsat, 0.0009 seconds, 0.0008513927459716797 seconds
```

Fig. 4. Simple Example Showing Verification Process.

SIMPLER [11]. Additionally, synthesis results for higher-input NOR gates (e.g., NOR5 and NOR6) show no significant differences in area or latency compared to NOR3 and NOR4 for the proposed approach.

B. Verification Results

Table II presents the verification results for the ISCAS'85 benchmark suite [17] using our proposed approach. The first column lists each circuit's name. The next twenty-one columns report the evaluation result (`S(SAT)` or `U(UNSAT)`), the number of clauses (N_{CL}), the number of variables (N_{Var}), the time taken by the parser to create the Verilog file from the micro-operations (t_P), the duration required to generate the Miter circuit and CNF clauses (t_M), the time spent verifying the CNF clauses with Z3 (t_{Z3}) and the overall verification time ($T_t = t_P + t_M + t_{Z3}$) for NOR2, NOR3, and NOR4 netlists. The following two columns show the number of clauses ($N_{CL(vS)}$) and verification time ($t_{(vS)}$) reported by veriSIMPLER [6] for NOR2 netlists. The last column shows the improvement of our verification method over veriSIMPLER [6], calculated as the ratio $t_{(vS)}/T_t$. All times are reported in seconds, with '-' indicating unavailable results for certain benchmarks.

The verification tool verifies the functional correctness of the proposed multi-input micro-operations across all ISCAS'85 benchmark functions for NOR2, NOR3, and NOR4 netlists. The verification results show a decrease in the number of clauses (N_{CL}) and variables (N_{Var}) when using NOR3 and NOR4 netlists compared to NOR2 netlists, for all benchmark functions. Specifically, NOR3 netlists result in an average reduction of 9.37% in clauses and 5.68% in variables, while NOR4 netlists achieve reductions of 5.12% in clauses and 1.37% in variables. However, the average runtime, which includes miter generation, parsing and verification with Z3, has slightly increased. The runtime for NOR3 is approximately 1.52× longer and for NOR4, it is about 1.06× longer compared to NOR2 netlists. The observed differences can be attributed to the trade-off between reduced clause and variable counts and a slight increase in runtime. NOR3 and NOR4 gates simplify the circuit by consolidating complex functions into a single gate, reducing clauses and variables, but slightly increasing runtime. This reduction simplifies the verification model, potentially easing solver demands. However, the higher fan-in of NOR3 and NOR4 gates slightly increases the complexity of miter generation, parsing and verification, leading to minor runtime increases compared to NOR2. Overall, the reduction in clauses and variables significantly enhances memory efficiency and scalability, making the slight runtime increase a reasonable trade-off. For a fair comparison, we only compare our verification results for NOR2 netlists with veriSIMPLER [6], which evaluates only NOR2. Table II highlights the runtime improvements for the NOR2 netlists achieved by our proposed method compared to veriSIMPLER [6]. Notably, results for four ISCAS'85 benchmarks (c1355, c6288, c2670, and c7552) are unavailable for veriSIMPLER [6]. For the remaining benchmarks, the proposed method demonstrates an average runtime improvement of approximately 6× for the NOR2 netlist.

This improvement is attributed to two key factors: (i) generating clauses after parsing the micro-operations into intermediate gate-level Verilog is more efficient than deriving them directly from the micro-operations, and (ii) while veriSIMPLER [6] processes each primary output (PO) iteratively, the proposed approach forms the complete miter first and generates CNF clauses for a single evaluation. Additionally, while veriSIMPLER [6] rely on [11] which overlooks the efficient synthesis of buffers and constants, the proposed method addresses these gaps by selecting benchmarks with specific features such as constants directly driving outputs, primary inputs exclusively driving outputs, and primary inputs acting both as NOR gate inputs and output drivers. These considerations strengthen the robustness of our synthesis method for multi-

TABLE I
Synthesis and Mapping Results for ISCAS'85 Benchmark

			Proposed Multi-input MAGIC Method												**SIMPLER**		**Improvement**							
Circuit			**NOR2**				**NOR3**				**NOR4**				**NOR2**		**NOR2 vs NOR3**		**NOR2 vs NOR4**		**SIMPLER vs NOR2**		
Name	**#PI**	**#PO**	**CBS**	L_T	L_{TA}	$S_t(s)$	**CBS**	L_T	L_{TA}	$S_t(s)$	**CBS**	L_T	L_{TA}	$S_t(s)$	CBS_S	L_{TS}	**CBS**	L_{TA}	**CBS**	L_{TA}	**CBS**	L_T	L_{TA}
c17	5	2	4x13	17	30	0.0004	4x13	17	30	0.0004	4x13	17	30	0.0004	-	-	1	1	1	1	-	-	
c432	36	7	17x118	158	503	0.003	10x96	125	289	0.002	13x90	112	260	0.001	62	237	2.08	1.74	1.71	1.93	0.03	1.5	0.4
c499	41	32	64x81	111	714	0.005	63x67	87	577	0.005	63x67	85	556	0.005	110	620	1.22	1.23	1.22	1.28	0.02	5.5	0.8
c880	60	26	39x95	126	664	0.004	44x97	123	499	0.004	51x75	137	475	0.003	142	512	0.86	1.33	0.96	1.39	0.03	4.06	0.7
c1355	41	32	64x83	113	700	0.005	63x70	91	593	0.005	63x67	85	556	0.005	111	619	1.20	1.18	1.25	1.25	0.02	5.4	0.8
c1908	33	35	51x127	169	792	0.005	39x102	130	569	0.005	39x114	139	535	0.004	122	588	1.62	1.39	1.45	1.48	0.01	3.4	0.7
c3540	50	22	68x146	195	1652	0.015	86x131	165	1169	0.010	91x142	172	1047	0.010	192	1434	0.88	1.41	0.76	1.57	0.01	7.3	0.8
c6288	32	32	58x358	477	2865	0.028	36x348	437	2410	0.022	36x373	461	2403	0.022	149	2938	1.65	1.18	1.54	1.19	0.007	6.2	1.04
c2670	233	32	75x83	110	1118	0.009	68x77	97	787	0.008	62x82	100	736	0.007	383	891	1.18	1.42	1.22	1.51	0.06	8.1	0.7
c5315	178	32	58x358	202	2428	0.028	178x103	130	1548	0.020	172x117	143	1501	0.020	351	2002	1.13	1.56	1.03	1.61	0.01	9.9	0.8
c7552	207	32	126x125	166	3006	0.026	119x156	197	1950	0.024	113x173	212	1824	0.022	535	2227	0.84	1.54	0.80	1.64	0.01	13.4	0.7
AVG	83	26	8742.45	167.63	1315.63	0.0116	7614.27	145.36	947.36	0.009	8094.72	151.18	902.09	0.009	215.7	1211.3	1.24	1.36	1.17	1.44	0.02	6.4	0.7

TABLE II
Verification Results for ISCAS'85 Benchmark

	Proposed Multi-input MAGIC Verification Method																					**veriSIMPLER**		**Improvement**
Circuit	**NOR2**							**NOR3**							**NOR4**							**NOR2**		**veriSIMPLER vs NOR2**
Name	***S/U***	N_{CL}	N_{Var}	t_P (s)	t_M (s)	t_{Z3} (s)	T_t (s)	***S/U***	N_{CL}	N_{Var}	t_P (s)	t_M (s)	t_{Z3} (s)	T_t (s)	***S/U***	N_{CL}	N_{Var}	t_P (s)	t_M (s)	t_{Z3} (s)	T_t (s)	$N_{CL(vS)}$	$T_t(vS)$ (s)	$t_{(vS)}/T_t$
c17	U	24	48	0.001	0.051	0.007	0.059	U	24	48	0.001	0.051	0.007	0.059	U	24	48	0.001	0.051	0.007	0.059	531	0.250	4.24
c432	U	477	1284	0.005	0.053	0.016	0.074	U	423	1163	0.005	0.080	0.009	0.094	U	481	1335	0.004	0.080	0.015	0.099	921	0.375	5.07
c499	U	962	2574	0.012	0.054	0.046	0.112	U	715	2075	0.013	0.080	0.065	0.158	U	786	2280	0.013	0.090	0.058	0.156	1738	1.829	16.33
c880	U	814	2109	0.010	0.054	0.056	0.120	U	676	1836	0.009	0.090	0.040	0.139	U	730	1987	0.008	0.070	0.068	0.146	1805	0.519	4.33
c1355	U	1070	2898	0.013	0.195	0.034	0.242	U	1027	2907	0.013	0.080	0.085	0.178	U	1098	3112	0.013	0.080	0.059	0.152	-	-	-
c1908	U	948	2598	0.012	0.056	0.127	0.195	U	762	2228	0.011	0.080	0.108	0.199	U	794	2319	0.011	0.080	0.076	0.167	2390	1.141	5.85
c3540	U	2166	6219	0.026	0.059	2.578	2.663	U	1982	5863	0.024	0.080	4.745	4.849	U	2114	6235	0.022	0.080	2.766	2.868	673	1.657	0.62
c6288	-	4816	14163	0.052	0.065	T.O.	-	-	5635	16730	0.070	0.090	T.O.	-	-	5635	16730	0.070	0.090	T.O.	-	-	-	-
c2670	U	1679	4299	0.029	0.057	0.032	0.118	U	1300	3515	0.027	0.080	0.032	0.139	U	1332	3598	0.025	0.080	0.031	0.136	-	-	-
c5315	U	3871	10512	0.112	0.065	0.342	0.519	U	3253	9335	0.107	0.090	0.417	0.614	U	3379	9700	0.106	0.090	0.353	0.549	6494	3.063	5.90
c7552	U	4450	12366	0.094	0.068	0.370	0.532	U	3486	10344	0.096	0.090	0.433	0.619	U	3694	10919	0.090	0.090	0.432	0.612	-	-	-
AVG	U	1934.27	5370	0.033	0.07	0.36	0.46	U	1753	5094.90	0.034	0.08	0.59	0.70	U	1824.27	5296.63	0.033	0.08	0.38	0.49	2650.28	1.18	6.04

input NOR-based MAGIC in-memory design. Verification also confirms the accuracy of the crossbar micro-operations.

V. Conclusion

In this paper, we present a comprehensive methodology for the synthesis, mapping, and verification of multi-input NOR-based MAGIC in-memory design on RRAM crossbars. Our approach introduces a standardized framework for crossbar micro-operations tailored to multi-input NOR logic, ensuring accurate mapping and efficient verification. Experimental results demonstrate the method's effectiveness, with NOR4 netlists outperforming NOR2 in both mapping latency and verification, achieving reduced clauses (N_{CL}) and variables (N_{Var}) with only a minor increase in runtime. These improvements enhance memory efficiency and scalability. Compared to existing methods, our approach achieves an average of 6× faster verification performance for NOR2 netlists while ensuring robust handling of buffers and constants, effectively addressing the limitations of previous techniques. In future work, we aim to optimize the verification process by reducing runtime and clause generation, extend the methodology to larger, fault-tolerant systems, and explore other in-memory logic styles.

Acknowledgement

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2).

References

[1] D. Strukov *et al.*, "The Missing Memristor Found," *Nature*, 2008.
[2] A. Sinha *et al.*, "Evolving nanoscale associative memories with memristors," in *IEEE NANO*, 2011.
[3] L. Xia *et al.*, "Technological exploration of RRAM crossbar array for matrix-vector multiplication," *Journal of Computer Science and Technology*, 2016.
[4] I. E. Ebong *et al.*, "CMOS and memristor-based neural network design for position detection," *Proceedings of the IEEE*, 2011.
[5] A. Deb *et al.*, "Automated Equivalence Checking Method for Majority based In-Memory Computing on ReRAM Crossbars," in *ASP-DAC*, 2023.
[6] C. Jha *et al.*, "veriSIMPLER: An Automated Formal Verification Methodology for SIMPLER MAGIC Design Style Based In-Memory Computing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.
[7] K. Akarvardar *et al.*, "Ultralow voltage crossbar nonvolatile memory based on energy-reversible NEM switches," *IEEE Electron Device Letters*, 2009.
[8] J. J. Yang *et al.*, "Memristive devices for computing," *Nature*, 2013.
[9] S. Kvatinsky *et al.*, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2014.
[10] P. Thangkhiew *et al.*, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *IDT*, 2016.
[11] R. Ben-Hur *et al.*, "SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
[12] S. Shirinzadeh *et al.*, "Logic Synthesis for RRAM-Based In-Memory Computing," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2018.
[13] D. Yadav *et al.*, "Look-ahead mapping of Boolean functions in memristive crossbar array," *Integration*, 2019.
[14] F. Lalchhandama *et al.*, "In-Memory Computing on Resistive RAM Systems Using Majority Operation," *Journal of Circuits, Systems and Computers*, 2022.
[15] S. Froehlich *et al.*, "Generation of Verified Programs for In-Memory Computing," in *DSD*, 2022.
[16] K. Qayyum *et al.*, "Exploring the potential of decision diagrams for efficient in-memory design verification," in *GLSVLSI*, 2024.
[17] M. Hansen *et al.*, "Unveiling the iscas-85 benchmarks: a case study in reverse engineering," *IEEE Design Test of Computers*, 1999.
[18] R. Brayton *et al.*, "ABC: An academic industrial-strength verification tool," in *ICCAD*, 2010.
[19] A. Biere *et al.*, "AIGER is a format, library and set of utilities for And-Inverter Graphs," available at https://github.com/arminbiere/aiger, 2024.
[20] L. d. Moura *et al.*, "Z3: An efficient SMT solver," in *TACAS*, 2008.