In-Memory SAT-Solver for Self-Verification of Programmable Memristive Architectures

Fatemeh Shirinzadeh[*], Arighna Deb[†], Saeideh Shirinzadeh[‡§], Abhoy Kole[‡], Kamalika Datta[*‡], Rolf Drechsler[*‡]
[*]Institute of Computer Science, University of Bremen, Germany
[†]Kalinga Institute of Industrial Technology, Bhubaneswar, India
[‡]German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
[§]Fraunhofer Institute for Systems and Innovation Research(ISI), Karlsruhe, Germany
{shirinfa, kdatta, drechsler}@uni-bremen.de, airghna.debfet@kiit.ac.in, {saeideh.shirinzadeh, abhoy.kole}@dfki.de

***Abstract*—Formal verification of programmable memristive architectures utilizing emerging nonvolatile memory technologies such as Resistive Random-Access Memory (RRAM) has only been recently addressed by a few works at the software level. In this paper we propose an in-memory SAT solver utilizing inherent analog features of RRAM that enables formal verification of arbitrary designs within resistive crossbars. More importantly, this allows self-verification of in-memory implementations as the correctness of designs can be dynamically checked. Additionally, the required architecture is presented, along with a complexity analysis for latency and hardware overheads.**

I. Introduction

Non-volatile memory technologies such as *Resistive RAM* (RRAM) possess advantages such as zero standby power, CMOS compatibility, fast switching, and ability to perform logic and analog computations. RRAM's computational capabilities have been used for neuromorphic computing and programmable logic-in-memory computing architectures as a potential solution to the memory wall issue in current computing systems. Various design methodologies have been explored to bring inherent capabilities of memristive devices into modern electronic systems. However, these approaches are mainly about logic synthesis, and verification has been out of focus despite being an important and complex stage in design flow. Recently, equivalence checking for logic-in-memory designs has been addressed at the software level performed on conventional computing systems [4], [6]. Nevertheless, no work has been reported to perform verification at the hardware level using in-memory structures. This enables self-verification which is of high importance for many complex applications. Self-verification is a process that integrates verification techniques into a system, enabling independent evaluation of its own functionality. This methodology assures the system's accuracy, correctness, and adherence to predefined specifications without relying on external verification [5].

In this paper, we propose for the first time an in-memory SAT-solver architecture that can be utilized for computation of any arbitrary SAT-instance for classical designs as well as for self-verification of in-memory designs directly performed on resistive crossbars. We propose two methods to compute the SAT-instance on a memristive crossbar utilizing analog features of RRAM devices that are being exploited in a logic context for the first time. The first method allows parallel evaluations in crossbar columns. The second method needs a single write cycle and performs evaluations sequentially to ensure error-free memory reads.

Experiments have been conducted for formal verification of majority-based in-memory design as a case study. Results report the required crossbar dimensions, as well as latency addressing both memories, writes, and the evaluation cycles that increases for larger functions. The proposed approach for the first time enables self-verification of memristive in-memory computing architectures. The scalability of the approach is therefore justifiable due to the cyclic nature of RRAM based computations.

II. Background

A. Basic Concepts

This section briefly presents the basic background required for making the paper self-contained.

1) Operations performed in RRAM: There are several universal logic primitives that can be executed within RRAM devices, viz. *Material Implication* (IMP), *Memristor-Aided LoGIC* (MAGIC), and *Resistive Majority Operation* (MAJ).

Besides the ability to execute aforementioned logical gates, RRAM possesses an analog computing capability that enables to perform *Multiply and Accumulate* (MAC) operation. MAC is already used in neuromorphic computing to speed up complex matrix multiplications. Assuming that the resistive values of the RRAM devices in the crossbar as depicted in Fig. 1, are initialized with $a_{j,k}^{-1}$, m MAC-operations can be conducted simultaneously within m crossbar columns by applying the voltages $x_1, \ldots, x_n$ to n to the rows. The outputs are the currents flowing in the corresponding crossbar columns, which is the sum of currents in each RRAM device, i.e. $i_j = \sum_{k=1}^{n} a_{j,k} \cdot x_k$. This can be denoted as $I = Ax$, where $I = (i_1, \ldots, i_m)^T$, $x = (x_1, \ldots, x_n)^T$ and the matrix A is defined as follows:

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

The m MAC-operations can be computed in parallel in a single cycle, each consisting of n multiplications. MAC-operation has been widely used for neuromorphic computing with RRAM; however, its capacity for logic domain has not

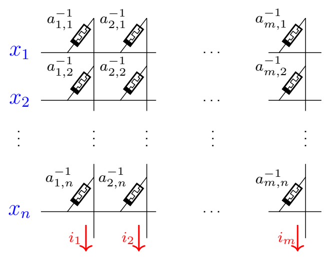


Fig. 1: MAC computation in RRAM crossbar

been explored yet. In this paper, we exploit MAC-operation for the first time for computation of logical functions in the context of our proposed in-memory SAT-solver.

2) SAT-solver: The Boolean Satisfiability Problem (SAT) is one of the classic problems in computer science. For a given Boolean formula, it attempts to determine whether a set of variable assignments exists such that the function is satisfied, i.e., is evaluated to TRUE. Whenever such an assignment exists, the function is considered to be satisfiable (sat) and otherwise unsatisfiable (unsat) [2]. Before a combinatorial problem can be solved by SAT methods, it must usually be encoded in *conjunctive normal form* (CNF). A CNF-clause is a conjunction of sub-clauses $\bigwedge_{i=1}^{n} c_i$, each sub-clause c_i being a disjunction of literals $\bigvee_{i=1}^{n} x_i$. Each literal x_i is either a Boolean variable v or its negation $\bar{v}$. In addition to its simplicity, CNF provides a common file representation and easy algorithm implementation.

In order to evaluate the clauses and sub-clauses of a CNF in a crossbar, initially the input variables are applied to the successive word lines as voltages corresponding to their logical values. As sub-clauses are disjunctions of variables, they can be computed directly by sensing the current flowing in each column as mentioned earlier in this section. Next, the current is measured and its equivalent voltage value is fed to the next allocated row (for the corresponding sub-clause) as an input to be used for further calculations. For clauses, which are conjunctions of sub-clauses (e.g, $a \wedge b$), De Morgan's law is used to convert the conjunction term into the negation of a disjunction (e.g, $\overline{\overline{a} \vee \overline{b}}$). This way, the complemented clause is evaluated in a single MAC-operation in a column. In this way, sub-clauses are computed in columns having all input variables allocated in rows. Similarly, the clauses are evaluated when the computed sub-clauses are allocated to available rows below the occupied rows in the crossbar.

B. Related Work

A compact, hardware-implemented Boolean Satisfiability (SAT) solver that can solve any arbitrary SAT-instance is presented in [10]. Recently, SAT-based equivalence checking methods for RRAM crossbar arrays are introduced in [4], [6]. More precisely, the equivalence checker proposed in [6] determines the functional equivalence between MIGs and a newly developed HDL-program supporting the RRAM operations. The work [4] introduces an automated equivalence checking methodology for majority-based in-memory designs,

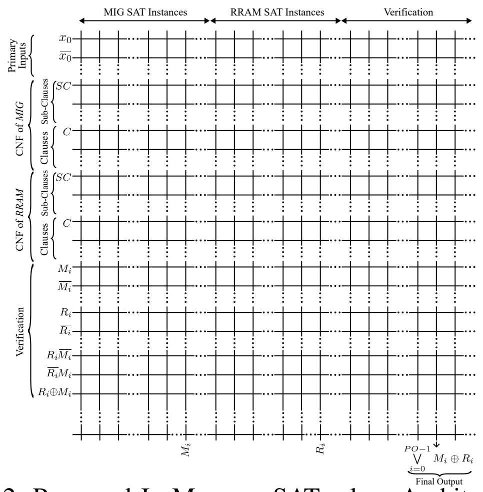


Fig. 2: Proposed In-Memory SAT-solver Architecture

which examines the functional equivalence or non-equivalence between the specification (e.g. MIGs) and the implementation (e.g. RRAM micro-operations). In this paper, we propose an in-memory SAT-solver implemented on RRAM crossbars that can perform self-verification of memristive architectures.

III. Proposed In-Memory SAT-Solver

Computation of CNF clauses is the first step to develop a SAT-solver within a memristive crossbar. The existing basic logic operations developed within RRAM cause various difficulties for full parallelism and suffer from inherent sequential nature, requiring many initialization or intermediate read cycles [3], [7], [9]. In this case, developing logic functions from higher complexity classes such as SAT-solvers based on such memristive logic primitives impose high costs in terms of latency, making them inferior as compared to software-based implementations. Thus, we propose to compute the SAT-instance based on MAC-operation (see Section II-A1), as it allows to exploit highly parallel computations utilizing the native analog features of RRAM devices.

A. Architectural Design

Initially, for both the MIG and the micro-operations, their respective CNFs (i.e. clauses and sub-clauses) M_i and R_i where $i = 0, \ldots, PO - 1$ are generated.

Fig. 2 shows the proposed architectural representation of the in-memory SAT-solver. The verification can be carried out in a single crossbar by allocating rows and columns in the following way:

$$\#Rows = \begin{cases} 2V + \sum_{i=1}^{L_{MIG}} (\sum SC + \sum 2C)_i + \\ \sum_{i=1}^{L_{RRAM}} (\sum SC + \sum 2C)_i + 2 & \text{if } PO = 1 \\ 2V + \sum_{i=1}^{L_{MIG}} (\sum SC + \sum 2C)_i + \\ \sum_{i=1}^{L_{RRAM}} (\sum SC + \sum 2C)_i + 3PO & \text{otherwise} \end{cases} \tag{1}$$

$$\#Cols = \begin{cases} \sum_{i=1}^{L_{MIG}} (\sum SC + \sum 2C)_i + \\ \sum_{i=1}^{L_{RRAM}} (\sum SC + \sum 2C)_i + 3 & \text{if } PO = 1 \\ \sum_{i=1}^{L_{MIG}} (\sum SC + \sum 2C)_i + \\ \sum_{i=1}^{L_{RRAM}} (\sum SC + \sum 2C)_i + 3PO + 1 & \text{otherwise} \end{cases} \tag{2}$$

where V, SC, C, and PO respectively denote the number of variables, the number of sub-clauses, the number of clauses, and the number of primary outputs. L_{MIG} and L_{RRAM} indicate the number of levels for MIG and RRAM-based designs, respectively. Since variables and their complements are used in sub-clauses, two rows will be occupied by each variable. Then at each level, each sub-clause occupies a row to generate the clause. Thereafter, clauses and their complements will occupy further rows.

Finally, to evaluate $M_i \oplus R_i$ for $i = 0, \ldots, PO-1$ in the crossbar, two more rows are required (for signals $M_i\bar{R}_i$, $\bar{M}_iR_i$) for each primary output. For multi-output function ($PO > 1$), the verification requires additional PO rows to compute

$$\bigvee_{i=0}^{PO-1} M_i \oplus R_i. \tag{3}$$

Similarly, each column will generate a sub-clause and clause according to the specifications of reference MIG and the RRAM-based design. Eventually, three more columns are required to generate $M_i\bar{R}_i$, $\bar{M}_iR_i$ and $M_i\bar{R}_i + \bar{M}_iR_i$ for each primary output and an additional column for the multi-output function to perform the Boolean OR operation as presented in Eqn.(3). Following is an example to clarify the process of realization of an arbitrary CNF function in the crossbar.

Example 1. Consider the CNF $f_2 = (f_1 + d).(\bar{f}_1 + \bar{a})$, where $f_1 = (a+b).\bar{c}$ and f_2 represents the i^{th} output (R) of the design under verification. Here the CNF is a two-level function with four primary inputs. To realize the CNF in a crossbar, initially the rows are assigned to variables and their complements (e.g. a and $\overline{a}$) as depicted in Fig. 3. Then, the first level sub-clause $a+b$ will be realized in a column and its inverted output is subsequently mapped in the next available row to generate the first level clause (f_1) in the complemented form as $\overline{(a+b)} + c$ in the next available column. Thereupon, two rows are used by the clause f_1 and its complement. Thereafter, second-level sub-clauses $f_1 + d$ and $\bar{f}_1 + \bar{a}$ are computed in the next two subsequent columns and their inverted outputs are placed in the next two rows in the crossbar to realize $\overline{f_2}$ in next column. For the verification step, (f_2, $\bar{f}_2$) and (M, $\bar{M}$) are placed in the next available rows in the crossbar to compute the inversion of $f_2\bar{M}$ and $\bar{f}_2M$ in two columns. The realization of the corresponding golden response (M) also takes place in a similar way. Finally, two more rows are allocated to $f_2\bar{M}$ and $\bar{f}_2M$ in order to compute the Boolean XOR operation.

B. MAC-based CNF computation on RRAM crossbar

Once rows and columns are assigned according to MIG and RRAM clauses, verification can be carried out through parallel or sequential execution of sub-clauses and clauses. The method chosen impacts cycle count, including crossbar column initialization and evaluation.

1) Method1 (Parallel MAC-Evaluation): This method requires to sequentially initialize crossbar columns and run MAC-operations to compute the disjunction sub-clauses, as

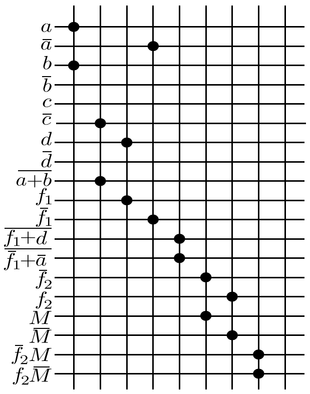


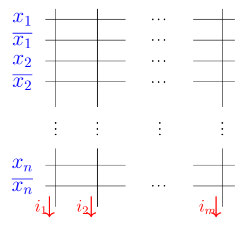
Fig. 3: Verification of an example CNF in the crossbar.

well as conjunctions representing CNF-clauses. Assume that the voltages applied to crossbar word lines in the equation describing MAC-operation (see Section II-A1) are set to values representing logical states of the input variables and their complements in consecutive rows as shown in Fig. 4a. Then the current flowing in each column can represent a CNF sub-clause or clause. This is performed by selectively switching RRAM devices in columns to on and off states, i.e. initializing their resistive states to low or high values, designating logic 1 and 0, respectively. For an arbitrary CNF sub-clause to be computed in i^{th} column, the RRAM devices representing the literals in the sub-clause are initialized with 1, while devices corresponding to absent literals are initialized with 0.

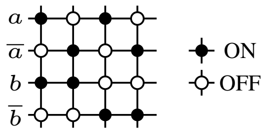
Example 2. Consider a 2-variable CNF expression with four sub-clauses $a+b$, $\bar{a}+b$, $a+\bar{b}$ and $\bar{a}+\bar{b}$. In order to realize a sub-clause $\bar{a}+b$ in the second column, the RRAM devices in the rows connected to $\bar{a}$ and b are initialized to 1, other devices are initialized to 0. Thus, it requires four initialization cycles for setting up the sub-clauses on four columns as shown in Fig. 4b.

As the initialization pattern required for this method is complex, each column can only be initialized independently. This requires $\#Cols$ (see Eqn. (2)) number of initialization cycles for the verification of MIG and RRAM clauses. In the evaluation stage, all sub-clauses can be evaluated by running MAC operations in parallel. In the next step, the conjunction of the sub-clauses is computed by applying De Morgan's law such that disjunction of complemented sub-clauses is evaluated by a single MAC-operation in a similar manner. For this purpose, current obtained in each column representing the computed sub-clauses is sensed and its inversion is fed back as voltages to further memory rows below those representing primary inputs. Then by allocating a new column and switching on the corresponding RRAM devices, a single MAC-cycle evaluates the negated conjunction, i.e. the CNF-clause. Then, the computed conjunction and its complement are driven to further rows below previous variables in the memory to be used for the computation of the next CNF-level. This procedure continues until the final CNF-level is computed.

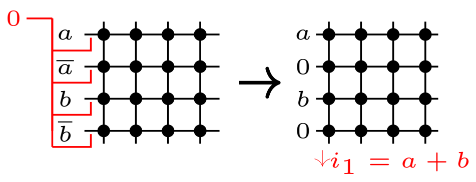
For the computation of SAT-instance, XOR of both CNFs representing the reference model (M_i) and the design to be



(a) Values applied to row for the realization of sub-clauses



(b) Realization of sub-clauses based on method 1



(c) Realization of sub-clauses based on method 2

Fig. 4: Required settings for in-memory SAT-solver for both presented methods.

verified (R_i) has to be computed for each output bit for $i = 0, \dots, PO-1$. The computation of each $\bar{M}_i R_i + M_i \bar{R}_i$ requires three columns to be initialized to perform three MAC-operations for each primary output (PO) for the two conjunctions and one disjunction. It may be noted that M_i and R_i can be computed on the same crossbar. Using this method, initialization cycles need to be performed sequentially for each of the CNFs, but their evaluation with MAC-operations can run in parallel. The number of MAC cycles required in the evaluation phase is:

$$\text{Evaluation Cycles} = \begin{cases} (L+1)2^{N+1} & \text{if } PO = 1 \\ (L+1.5)2^{N+1} & \text{otherwise} \end{cases} \tag{4}$$

where L and N denote the maximum number of levels and variables present in both CNFs representing the MIG and RRAM-based design and PO indicates the number of primary outputs. The number of cycles required for computation of an example SAT-instance is explained below.

Example 3. To verify a SAT instance consisting of a CNF representing a MIG expression with 5 sub-clauses and single level and an RRAM-based design expression with 4 sub-clauses and single level. If CNF expressions consist of five Boolean variables describing a function with a single primary output, then the number of cycles required for verification is 142.

2) Method2 (Sequential MAC-Evaluation): Method1 enables fast parallel MAC-operations, yet long columns can lead to erroneous reads. Accumulated currents from numerous low conductivity devices might mistakenly evaluate as 1 instead of the correct value of 0. Moreover, *Method1* might require many initialization cycles for larger CNF representations that are not desired due to the low write endurance of RRAM devices.

Therefore, we propose a second method that needs only a single initialization cycle to switch devices to high conductivity states. It performs MAC-operations by controlling row voltages, allowing a single MAC-operation per cycle.

Assuming that all RRAM devices are switched on, a given sub-clause can be evaluated in an arbitrary column by providing the inverted or non-inverted forms of primary inputs or previously computed CNF-clauses that are present as variables in the sub-clause at the crossbar rows as shown in Fig. 4a. In this case, all inputs and intermediate variables (CNF-clauses) are applied to certain rows as in the previous method. However, by use of this method, we need a more complex row-driver that can selectively set a row to 0 if the variable applied is missing in the sub-clause or clause under computation. From the first CNF-level, sub-clauses are computed independently in columns using MAC cycles. This resembles the prior method, with MAC-results applied to rows for conjunction. This repeats like *Method1*, but without extra initialization, and with sequential MAC-operations as illustrated below.

Example 4. Consider again the 2-variable CNF expression with four sub-clauses $a+b$, $\bar{a}+b$, $a+\bar{b}$ and $\bar{a}+\bar{b}$. It requires a single initialization cycle and four MAC (evaluation) cycles for the execution of all the sub-clauses using *Method2* as shown in Fig. 4c.

Following this method, if CNFs in the SAT-instances are on the same crossbar, they are sequentially computed for each MAC-operation. Then we perform XOR for MAC evaluation using *Method1*, but this approach requires just one initialization cycle for the entire computation. The number of MAC cycles required for evaluation is:

$$\text{Evaluation Cycles} = \#Cols \times 2^N \tag{5}$$

where $\#Cols$ can be evaluated using Eqn. 2 and N denotes the number of variables present in MIG and RRAM clauses.

C. Complexity Analysis

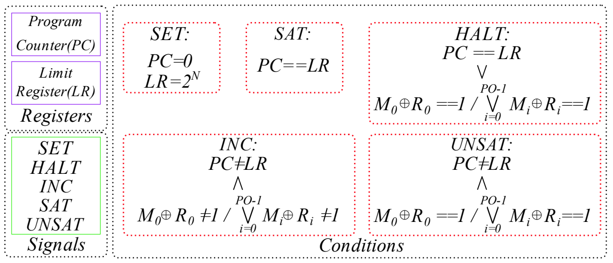
The verification process is carried out in two phases: initialization and evaluation. Using *Method1: Parallel MAC-Evaluation*, the number of cycles required for verification is:

$$\text{Verification Cycles}_{\textit{Method1}} = \\ \#Cols + \begin{cases} (L+1)2^{N+1} & \text{if } PO = 1 \\ (L+1.5)2^{N+1} & \text{otherwise} \end{cases} \tag{6}$$

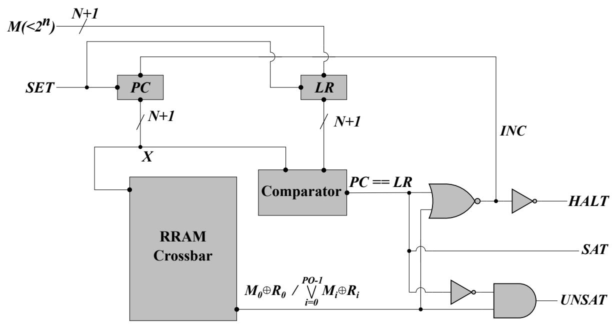
Similarly, The cycles required by the *Method2: Sequential MAC-Evaluation* process is:

$$\text{Verification Cycles}_{\textit{Method2}} = 1 + \#Cols \times 2^N \tag{7}$$

Since the maximum number of CNF-levels for both MIG and RRAM-based design expressions is less than the sum of clauses and sub-clauses, i.e. $L < SC + C$, *Method2* requires much higher evaluation cycles due to the presence of the factor 2^N in Eqns. (4) and (5). It further increases for functions with multiple primary outputs, as $\#Cols$ in Eqn. (6) and (7) is increased by a factor $3PO$ (see Eqn. 2). There is a tradeoff between the usage of the verification approaches.


Program Counter(PC)
Limit Register(LR)
Registers
SET
HALT
INC
SAT
UNSAT
Signals
SET: PC=0 LR=2^N
SAT: PC==LR
HALT: PC == LR ∨ M0⊕R0 ==1 / ∨ i=0 PO-1 Mi⊕Ri==1
INC: PC≠LR ∧ M0⊕R0 ≠1 / ∨ i=0 PO-1 Mi⊕Ri ≠1
UNSAT: PC≠LR ∧ M0⊕R0 ==1 / ∨ i=0 PO-1 Mi⊕Ri==1
Conditions

(a) Registers and Control Signals



(b) Data Flow Path of the In-Memory SAT-solver

Fig. 5: Design Implementation

Hence, *Method1* requires a higher number of initialization cycles, i.e. $\mathcal{O}(\#Cols)$, but the alternate approach suffers from higher evaluation cycles, i.e. $\mathcal{O}(\#Cols \times 2^N)$.

D. Design Implementation

Fig. 5a shows the registers, control signals, and the conditions associated with those signals. Fig. 5b shows the data flow path of the in-memory SAT-solver with various registers and control signals. These are the salient features of the in-memory SAT architecture for verifying an N-bit function:

1) Program Counter (PC), stores the current value of evaluation which is between 0 to 2^{N-1}. It is a $N+1$ bit register and is initialized to 0. After every step, it is incremented by 1.
2) Limit Register (LR), stores the value which is the maximum number of possible combinations for N-bit, i.e. 2^N. This is initialized to 2^N at the start of the evaluation.
3) SET signal initializes the PC and LR registers for evaluation.
4) HALT signal is generated when either PC has reached LR or the results of the two functions for certain input combinations are different.
5) INC signal is used to increment PC if $PC \neq LR$ and $M_0 \oplus R_0 \neq 1$ when $PO = 1$ or $\bigvee_{i=0}^{PO-1} M_i \oplus R_i \neq 1$ when $PO > 1$.
6) UNSAT signal become active if $PC \neq LR$ and $M_0 \oplus R_0 = 1$ for $PO = 1$ or $\bigvee_{i=0}^{PO-1} M_i \oplus R_i = 1$ for $PO > 1$. This means when either the SAT-solver is still checking for all possible inputs and for some inputs $M_0 \oplus R_0$ or $\bigvee_{i=0}^{PO-1} M_i \oplus R_i$ is 1.
7) SAT signal is generated when PC value has reached the content of LR register.
8) HALT signal is generated when either $PC = LR$ i.e. PC had reached its maximum value or $M_0 \oplus R_0 = 1$ for $PO = 1$ or $\bigvee_{i=0}^{PO-1} M_i \oplus R_i = 1$ for $PO > 1$ i.e. for some input combinations the output of the golden response and the design under verification does not match.

IV. Experimental Results

To evaluate the in-memory SAT-solver architecture proposed in this work, we have considered IWLS and ISCAS-85 benchmarks [1], [8]. As a case study, we have considered the equivalence checking approach reported in [4] that finds the equivalence between the MIG function representations of the benchmarks and their corresponding micro-operations executed on the RRAM crossbar arrays. To execute the complete equivalence checking approach [4] on our proposed architecture, we have utilized the SAT-instances obtained from the MIGs and micro-operations. Note that any SAT instances consisting of multiple CNF clauses can be executed in the proposed architecture. That is, the architecture is not restricted to only the equivalence checking of MIGs and RRAM micro-operations, but can be utilized for any SAT-based equivalence checking method.

TABLE I: Details of the benchmarks

Benchmark		MIGs			Micro-operations		
Name	PI/PO	#C	#SC	#L	#C	#SC	#L
exam1	3/1	6	18	4	9	27	9
exam3	4/1	10	30	4	16	48	9
xor5	5/1	6	18	4	19	57	11
con1	7/1	8	24	4	12	36	8
con2	7/1	9	27	4	13	39	8
newtag	8/1	9	27	4	13	39	8
newill	8/1	20	60	8	28	84	10
9sym	9/1	25	75	14	34	102	26
max46	9/1	132	396	12	178	534	18
sym10	10/1	35	105	17	45	135	29
t481	16/1	25	75	5	36	108	11
rd32	3/2	3	9	2	6	18	3
rd53	5/3	9	27	5	15	45	9
rd73	7/3	12	36	8	16	48	12
rd84	8/4	21	63	10	28	84	21
c6288	32/32	1867	5601	116	2347	7041	136
c1908	33/25	296	888	28	388	1164	41
c432	36/7	95	285	23	124	372	37
c499	41/32	292	876	21	356	1068	28
c3540	50/22	824	2472	33	1159	3477	44

Table I shows the details of the benchmarks. The first column provides the name of the benchmark, the number of primary inputs (PI) and the number of primary outputs (PO). The second and third columns report the number of clauses (#C), number of sub-clauses (#SC), and number of levels (#L) in the SAT-instances of the MIGs and corresponding micro-operations, respectively.

Table II summarizes the results. The first and second columns show the benchmark name and crossbar array size ($r \times C$), where r and C denote the number of rows and columns, respectively in the crossbar array. The third and fourth columns respectively report the number of write cycles

TABLE II: Experimental results for single and multi-output functions

Benchmark	Crossbar	write cycles		MAC cycles		Total cycles	
Name	$r \times c$	Method 1	Method 2	Method 1	Method 2	Method 1	Method 2
exam1	68×63	63	1	160	504	223	505
exam3	114×107	107	1	320	1712	427	1713
xor5	112×103	103	1	768	3296	871	3297
con1	96×83	83	1	2304	10624	2387	10625
con2	104×91	91	1	2304	11648	2395	11649
newtag	106×91	91	1	4608	23296	4699	23297
newill	210×195	195	1	5632	49920	5827	49921
9sym	256×239	239	1	27648	122368	27887	122369
max46	1260×1243	1243	1	19456	636416	20699	636417
sym10	342×323	323	1	61440	330752	61763	330753
t481	278×247	247	1	1572864	16187392	1573111	16187393
rd32	48×43	43	1	72	344	115	345
rd53	115×106	106	1	672	3392	778	3393
rd73	135×122	122	1	3456	15616	3578	15617
rd84	224×209	209	1	11520	53504	11729	53505
c6288	17016×16953	16953	1	1.18×10^{12}	7.28×10^{13}	1.18×10^{12}	7.28×10^{13}
c1908	2877×2812	2812	1	7.30×10^{11}	2.42×10^{13}	7.30×10^{11}	2.42×10^{13}
c432	969×898	898	1	5.29×10^{12}	6.17×10^{13}	5.29×10^{12}	6.17×10^{13}
c499	2770×2689	2689	1	1.29×10^{12}	5.91×10^{15}	1.29×10^{12}	5.91×10^{15}
c3540	8098×7999	7999	1	1.02×10^{17}	9.01×10^{18}	1.02×10^{17}	9.01×10^{18}

and the number of MAC cycles required for both methods to implement the equivalence checking using our proposed architecture. The summation of the write cycles and the MAC cycles, i.e. the total cycles required for evaluating the equivalence between MIGs and RRAM micro-operations based on *Method1* and *Method2* are reported in the final column. The results demonstrate that the SAT-instances obtained from the MIGs and from the micro-operations strongly influence the size of the crossbar array, while the number of primary inputs and primary outputs of the functions very loosely affect the size of the crossbar. That is, higher the number of clauses, larger will be the size of the crossbar. Considering the performance of the architecture, the equivalence checking can be executed in two ways on the architecture: *parallel execution (Method1)* and *sequential execution (Method2)*. Depending on the inputs and CNF clause levels, the runtime will vary. As evident from Table II, the total cycles required to execute the equivalence checking using *Method2* is almost 10 times more than that of *Method1*, since *Method1* allows parallel execution of the SAT-instances. However, the number of write cycles remains constant in the case of *Method2* as opposed to *Method1* where the write cycle varies with the number of SAT clauses. The constant write cycle makes *Method2* advantageous over *Method1* in terms of the memristor device lifetime which becomes a limiting factor after performing repetitive write operations. Overall, the experimental evaluation confirms that our proposed architecture can efficiently handle any large SAT instances that can be mapped to a crossbar consisting of more than 1000 rows and columns.

V. Conclusions

This paper presents an in-memory SAT-solver architecture for the first time, that utilizes the inherent computational capabilities of RRAM devices. The proposed SAT-solver can compute any arbitrary SAT-instance and more importantly, can be used for dynamic self-verification within in-memory computing architectures. Two methods have been proposed for parallel and sequential evaluation of a given SAT-instance. The latter aims to ensure error-free memory reads at the expense of a higher number of evaluation cycles. Time and area complexities have been addressed for both methods and experiments have been performed for equivalence checking of majority-based logic-in-memory designs as a case study. Being a pioneering approach, no comparisons with other approaches are available.

Acknowledgement

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2 and SH 1917/1-2).

References

[1] C. Albrecht. Iwls 2005 benchmarks. Technical report, June 2005.

[2] A. Biere, M. Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[3] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams. 'memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464(7290):873–876, 2010.

[4] A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler. Automated equivalence checking method for majority based in-memory computing on reram crossbars. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 19–25, 2023.

[5] R. Drechsler, H. M. Le, and M. Soeken. Self-verification as the key technology for next generation electronic systems. In *2014 27th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–4, 2014.

[6] S. Froehlich and R. Drechsler. Generation of verified programs for in-memory computing. In *Digital System Design (DSD-2022)*, 2022.

[7] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. The programmable logic-in-memory (plim) computer. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 427–432. Ieee, 2016.

[8] M.C. Hansen, H. Yalcin, and J.P. Hayes. Unveiling the iscas-85 benchmarks: a case study in reverse engineering. *IEEE Design Test of Computers*, 16(3):72–80, 1999.

[9] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

[10] B. Ustaoglu, S. Huhn, D. Große, and R. Drechsler. Sat-lancer: a hardware sat-solver for self-verification. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 479–482, 2018.