Equivalence Checking for Flow-Based Computing using Iterative SAT Solving

Sven Thijssen[1], Muhammad Rashedul Haq Rashed[2], Md Rubel Ahmed[3], Suraj Singireddy[4], Sumit Kumar Jha[5], Rickard Ewetz[6]
[1]Department of Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL, USA
[2]Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA
[3]Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL, USA
[4]Department of Electrical and Computer Engineering, University of Texas at San Antonio, San Antonio, TX, USA
[5]Knight Foundation School of Computing and Information Sciences, Florida International University, Miami, FL, USA
[6]Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
sthijssen@fau.edu,muhammad.rashed@uta.edu,mdrubel.ahmed@ucf.edu,suraj.singireddy@utsa.edu,sumit.jha@fiu.edu,rewetz@ufl.edu

Abstract

Processing in-memory is projected to shatter the von Neumann bottleneck and enable acceleration of data-intensive applications. Flow-based computing is an efficient in-memory computing paradigm for accelerating the execution of Boolean logic. While recent synthesis algorithms can map complex functions into flow-based computing circuits, the functional correctness cannot be verified using state-of-the-art equivalence checking techniques. The challenge is that non-volatile memory devices are intrinsically bi-directional, which introduces cycles in the computational graph. These cycles break traditional equivalence checking methods that are based on SAT formulations. In this paper, we propose a framework for equivalence checking of flow-based computing circuits that is called FlowSAT. The framework captures each circuit using an undirected computational graph. The key idea of FlowSAT is to introduce helper variables, in the form of arrows, that dynamically convert the undirected graph into a directed graph. This facilitates equivalence checking to be performed using traditional SAT formulations. However, it is prohibitively expensive to ban all possible cycles using arrow variables. Therefore, we propose to eliminate cycles by iteratively adding constraints to the SAT formulation. Our experimental evaluation demonstrates that FlowSAT is up to an order of magnitude faster than state-of-the-art methods. The framework is capable of verifying all 20/20 benchmark circuits, while the previous state-of-the-art technique is only capable of verifying 12/20 circuits within a time limit of one hour.

ACM Reference Format:
Sven Thijssen, Muhammad Rashedul Haq Rashed, Md Rubel Ahmed, Suraj Singireddy, Sumit Kumar Jha, Rickard Ewetz. 2024. Equivalence Checking for Flow-Based Computing using Iterative SAT Solving. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676721

This work was in part supported by NSF #2319399 and NSF #2404036. The research was conducted while Sven Thijssen, Muhammed Rashedul Haq Rashed, and Rickard Ewetz were affiliated with the University of Central Florida.

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676721

1 Introduction

With the advent of data-centered applications, the slow-down of Moore's law [41], and the end of Dennard scaling [13, 45], there have been substantial investments in emerging technologies and alternative computing paradigms [1, 21, 29, 48], such as in-memory computing. Processing in-memory (PIM) can be realized using traditional CMOS devices or emerging non-volatile memory devices such as resistive random-access memory (ReRAM) [47], spin-transfer torque magnetic random-access memory (STT-MRAM) [18], phase change memory (PCM) [7], and ferroelectric field effect transistors (FeFET) [2]. In addition to minimizing standby power consumption, the non-volatile memory devices facilitate energy-efficient analog in-memory computation. Moreover, in-memory computing systems perform computation within the memory, which paves the way to breaking the Von-Neumman barrier and unleashing orders of magnitude (up to three) improvements in energy-efficiency [19, 27].

Many logic styles for PIM have been proposed, including solutions based on analog matrix-vector multiplication [17], Bit-wise-in-bulk [28], material implication logic (IMPLY) [5], memristor-aided logic (MAGIC) [26], majority-based logic (MAJORITY) [14], path-based computing (PATH) [38], and flow-based computing (FLOW) [23]. While analog in-memory computing has appealing properties for accelerating applications that can tolerate low-precision (e.g. neural networks and search), many other applications require digital in-memory computing to provide guarantees on the computational accuracy [32]. Several contenders have emerged for accelerating Boolean logic using in-memory computing [5, 26, 38]. Among these, flow-based computing has demonstrated immense potential due to the fast and efficient computations [36].

Research on flow-based computing has primarily focused on mapping a Boolean function ϕ to a crossbar design $\mathcal{D}$. Solutions include approaches such as model counting [9], conjunctive normal form [43], and binary decision diagrams (BDDs) [8]. However, equally important is the problem of verifying the functional correctness using equivalence checking. Equivalence checking is used to ensure that no errors are introduced within logical optimization techniques. Equivalence checking is also a critical tool for test and debugging [34]. Equivalence checking for traditional CMOS circuits is performed using SAT solving [30], as follows: a miter circuit is first formed using the designed circuit $\mathcal{D}$ and the specification. Next, a SAT problem is formulated and solved to identify any inputs that give different outputs from the design and specification.

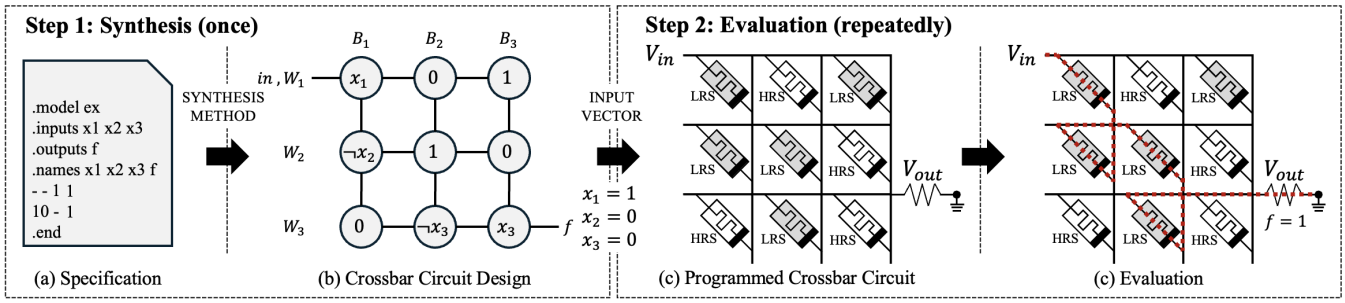


Figure 1: Boolean functions can be executed using flow-based computing using a synthesis and evaluation step. (a) In the first step, the input is a specification in, for example, BLIF file format, is (b) compiled into a crossbar design [36]. In the second step, (c) the non-volatile memory devices are programmed to high or low resistive states, depending on the truth values of the input variables (x_1, x_2, x_3). (d) Finally, the Boolean function (f) is evaluated by applying a voltage at the input and grounding the output. Subsequently, an electrical current *flows* through the programmed crossbar circuit. When the electrical current reaches the output, the Boolean function evaluates to true.

The approach is fast and efficient and leverages that the computational graph of CMOS circuits is acyclic [30]. Unfortunately, the equivalence checking problem for flow-based computing is significantly more challenging due to the fact that non-volatile memory devices are bidirectional. This results in the underlying computational graph being undirected, which allows cycles to be formed in the graph. These cycles break standard equivalence checking approaches based on traditional SAT formulations.

Early investigations on equivalence checking for flow-based computing are based on brute-force enumeration [10], graph reachability [37], recurrent neural networks (RNNs) [34]. These methods only scale to crossbar designs of modest size. Mainly due to the exponential time complexity with respect to the number of input variables. A more recent work performs equivalence checking using bounded model checking (BMC) [40]. That work decoupled the undirected graph into a directed graph by replicating the crossbar $(N + M)$ times, which enables equivalence checking methods to be scaled up with one to two orders of magnitude [40]. Unfortunately, the size of the SAT formulation grows rapidly with the size of the crossbar, making it unable to verify large crossbar designs.

In this paper, we propose a framework for equivalence checking of flow-based computing circuits called FlowSAT. The key idea of the framework is to introduce an arrow variable that specifies the direction of each edge in the underlying computational graph. The arrow variables effectively convert the undirected graph into a directed graph without replicating the crossbar design as in [40]. This results in a substantially more concise SAT formulation. The arrow variables specify the direction of the current flow with respect to a particular input vector. To eliminate current flowing in cycles within the crossbar, constraints using the arrows must be added to ban the cycles. However, it is prohibitively expensive to enumerate all possible cycles within a crossbar design [12]. Moreover, many cycles will, in reality, never occur due to conflicting Boolean variables. For example, a cycle containing a Boolean variable x and its complement $\neg x$ will never be formed. Therefore, in the FlowSAT framework, we propose to iteratively and incrementally add constraints that eliminate cycles. This results in a SAT formulation with $O(M+N+K)$ variables, where M, N, K, are the number of wordlines, bitlines, and memristors with a Boolean variable assigned, respectively. This is substantially fewer variables than the state-of-the-art method XSAT, which requires $O((M+N)\cdot(NM)+K)$ variables. For our experimental evaluation, we compare our proposed FlowSAT framework with enumeration [10], RNN [34], CHECK [37], and XSAT [40] on 20 benchmarks. This comparison is performed for both the equivalent case and the nonequivalent case. From the evaluation, we conclude that FlowSAT is an order of magnitude faster than the state-of-the-art framework XSAT. Moreover, FlowSAT is the only framework that is able to prove equivalence for all 20 benchmark circuits. Furthermore, we analyze the effects of cycles on the behavior of our proposed iterative SAT formulation.

The paper is organized as follows: preliminaries are provided in Section 2, where we discuss flow-based computing and equivalence checking. The FlowSAT framework is introduced in Section 3. The experimental evaluation is discussed in Section 4. The paper is concluded in Section 5.

2 Preliminaries

2.1 Flow-Based Computing

Flow-based computing [31, 39] is a digital in-memory computing paradigm on crossbar arrays. A crossbar is a mesh of two or multiple layers of conducting nanowires, as shown in Fig. 1(c). Each layer is perpendicular to its adjacent layers, and nanowires in a layer are parallel to one another. Between two consecutive layers, NVM devices connect nanowires at the intersections. As shown in Fig. 1, evaluation of Boolean functions is performed using two steps: (1) a synthesis step and (2) an evaluation step.

Step 1: Synthesis. In this step, a Boolean function (specification) is provided as input, as shown in Figure 1(a). The specification may be in BLIF, Verilog, or PLA file format, which is subsequently synthesized into a crossbar circuit design, as shown in Figure 1(b). Here, $f = (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee x_3$ is the specification. In the crossbar circuit design, the input variables ($\{x_1, x_2, x_3\}$), the negation of the input variables ($\{\neg x_1, \neg x_2, \neg x_3\}$), and the truth values ($\{0, 1\}$) are assigned to the non-volatile memory devices, as well as the wordlines for the input and output. A wide variety of synthesis techniques have been proposed to construct a crossbar circuit design for a given Boolean function. These techniques have been conjunctive normal forms (CNFs) [43], negation normal forms (NNFs) [23], free binary decision diagrams (FBDDs) [16], and reduced ordered binary decision diagrams (ROBDDs) [8].

Step 2: Evaluation. In this step, the Boolean function is evaluated for an assignment of truth values to the input variables ($\{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 0\}$. Based on the truth values, the non-volatile memory devices are programmed to high or low resistive states (HRS/LRS) for a logical '0' or '1', respectively. In Figure 1(c), the state of the programmed crossbar circuit is shown, where the NVM devices are either in LRS (dark gray) or HRS (white). The last sub-step, the Boolean function, is evaluated by applying a voltage at the input and grounding the output wordline. Consequently, an electrical current flows through the nanowires and NVM devices of the crossbar array, as shown in Figure 1(d). When an electrical current is (not) measured at the output, the Boolean function evaluates to true (false).

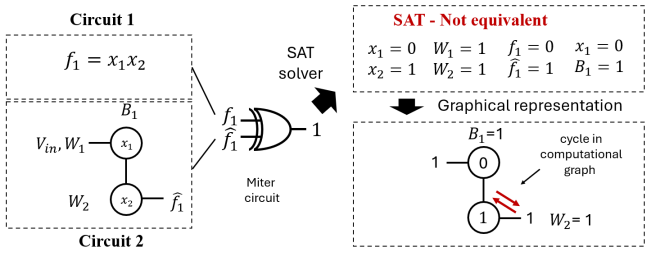


Figure 2: Example of equivalence checking flow using SAT. (a) The miter circuit of an $f_1 = x_1 \wedge x_2$ from specification and using a flow-based computing circuit. (b) The solution from the SAT solver both in terms of variables and using a graphical realization. The SAT solver determines that the circuits are non-equivalent. Graphically, it can be observed that bitline B_1 drives wordline W_2 and vice versa. This situation could not occur in a circuit with directed gates.

2.2 Equivalence Checking

Equivalence checking is the problem of verifying the functional equivalence between a circuit design and a specification (also referred to as "golden model"). Even though this is an NP-hard problem, over the years, many techniques have been proposed, resulting in hybrid solutions. For example, techniques for equivalence checking may include one or more of the following: simulation, synthesis, and SAT solving [30].

Simulation may be an effective way to show non-equivalence between two circuits. When the input space is small, simulation may be used to show equivalence [25]. However, when the input space is large, simulation may not be a scalable method and can be used in combination with formal methods. Synthesis can be leveraged for equivalence checking. For example, given a variable ordering, binary decision diagrams are canonical representations for a Boolean function. By compiling both the design and specification into a BDD, the two implementations can easily be compared. Further, and-inverter graphs (AIGs) have been employed as intermediate data structures to reduce the overall problem size [6]. Lastly, one of the most successful methods for determining (non-) equivalence between two circuits is formulating the problem as a satisfiability (SAT) problem [12, 35]. To formulate the problem as a SAT problem, the two circuits are bound together by constructing a miter circuit. For the miter, the same output variables in both circuits are joined together using an XOR gate, which is subsequently joined together using an OR gate (disjoined). Then, the circuit is converted into conjunctive normal form (CNF). CNF is a format for a Boolean expression that consists of a conjunction (AND) of clauses, where a clause is a disjunction (OR) of Boolean literals. The CNF representation of a circuit can directly be obtained using Tseitin's transformation [42]. Many conversions, such as Tseitin's transformation, have been proposed to obtain a CNF formulation for a Boolean formula. Lastly, the CNF formulation is fed to a SAT solver [35]. When the SAT problem is UNSAT, the circuits are equivalent because no assignment of the input variables can be found such that the output variables of both circuits are different. On the other hand, when the SAT problem results in SAT, such assignment can be found and both circuits are non-equivalent. SAT solvers have been fine-tuned over the years with the objective of reducing the overall runtime.

2.3 SAT Formulations and Undirected Graphs

In this section, we outline the challenge of performing equivalence checking for circuits with bi-directional devices. We exemplify the challenge using a minimal example in Figure 2.

The figure shows a specification of the Boolean functions $f = x_1 \wedge x_2$. A correct realization of the circuit using crossbar design $\mathcal{D}$ for flow-based computing is shown in Figure 2(a). Both the specification and the crossbar design are connected to an XOR gate to form a miter circuit. Next, the miter circuit is translated into a CNF representation using Tseitin's transformation. It is straightforward to translate the AND-gate and XOR gate into CNF form. To translate crossbar design into CNF, a Boolean variable is created for each wordline (W_1,W_2) and bitline (B_1). Next, two clauses are added per memristor to capture their functional behavior (details in the next section). The clauses ensure that the wordline and bitline have the same value if the memristor connecting the two are turned on. We also introduce a constraint that ensures that a wordline (or bitline) can only be true if at least one bitline connected to the wordline is true and the corresponding memristor is true. The outlined formulation is fed to a SAT solver. As shown in the figure, the SAT solver determines that the formulation is SAT and that the two circuits are not equivalent. This is problematic because the crossbar design $\mathcal{D}$ is a correct realization of the f. The error stems from the fact that when $x_1 = 0$ and $x_2 = 1$, the output should be equal to 0. However, the example shows that bitline B_1 is 1 and wordline W_2 is 1, but neither is connected to the input. Essentially, the bitline-world line pair is driving itself using a cycle. The SAT solver can form the cycles since the underlying computational graph is undirected due to the bidirectional devices. However, there is no physical realization of such cycles. Moreover, the crossbar cannot easily be made directed since the flow of current across a memristor may change for different inputs.

This problem was solved by replicating the crossbar $(N + M)$ times using Bounded model checking in XSAT [40]. The replication (or time unrolling) converts the undirected graph into a directed graph at the expense of making the SAT formulation have a factor of $(N+M)$ more variables. In this work, we instead introduce arrow variables that convert the undirected graph into an acyclic graph for each input vector, i.e., the acyclic graph defined by the arrow variables may change for each input.

3 The FlowSAT Framework

In this section, we introduce the FlowSAT framework. The input to the framework is a specification of a function ϕ and a crossbar design $\mathcal{D}$. The objective is to verify if the specification and the crossbar design are equivalent. The output from the framework is equivalent or not equivalent. An overview of the framework is shown in Figure 3.

Our proposed solution is based on introducing an arrow helper variable for each memristor in the crossbar. The arrow variables designate the direction of the current flowing through each memristor device. Using the arrows, it is straightforward to ensure that a bitline-wordline pair cannot drive each other, which was the root cause of the failure scenario in Figure 2. We present a SAT formulation that solves equivalence checking problem for cycles of maximum length of 2 in Section 3.1. The length of a cycle is defined as the number of memristor devices that must be traversed to return to the origin. In Section 3.2, we extend the framework to handle cycles of length $L > 2$. The two cases are handled differently because the arrow variables directly eliminate all cycles of length 2. However, explicit constraints are required to be added to ban cycles of longer length. Nevertheless, it is prohibitively expensive to enumerate and ban all possible cycles in an undirected graph [12]. Fortunately, it is not necessary to ban all cycles because most cycles cannot occur in reality. For example, current cannot flow through a cycle containing a Boolean variable x and its complement $\neg x$. Therefore, we propose to instead analyze the solution from the SAT solver and incrementally and iteratively ban cycles, which is shown at the bottom of Figure 3.

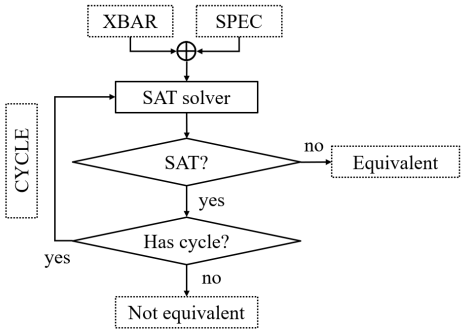


Figure 3: Overview of the FlowSAT framework.

3.1 Base Formulation

The base formulation consists of three types of constraints: (1) memristor constraints, (2) driver constraints, and (3) arrow constraints. Each of the constraints is discussed further in subsequent sections.

3.1.1 Variables. Given a crossbar circuit with dimensions $M \times N$, we introduce variables for both the M wordlines and the N bitlines. These variables are $W_i, 1 \leq i \leq M$ and $B_i, 1 \leq i \leq N$. This is a total of $(N + M)$ variables for a NxM crossbar.

3.1.2 Memristor Constraints. For each memristor $M_{i,j}, 1 \leq i \leq M, 1 \leq j \leq N$, we introduce a constraint that models the behavior of the intersection at the wordline W_i and bitline B_j. Given an electrical current on a wordline W_i (bitline B_j), then the electrical current can only dissipate on the bitline B_j (wordline W_i) if the memristor $M_{i,j}$ at its intersection is in a low resistive state. Otherwise, if there is no electrical current on wordline W_i (bitline B_j), then the state of the bitline B_j (wordline W_i) is inconclusive. The bitline may carry an electrical current by means of another wordline, or may not carry an electrical current. For the SAT formulation, we must ban the *invalid* constraints. In this case, an invalid constraint solution exists when the wordline W_i carries an electrical current, memristor $M_{i,j}$ is in a low resistive state, and the bitline B_j has no electrical current.

To ban these invalid solutions, we add two clauses to the SAT formulation, one for each invalid solution. These constraints are:

$$(W_i \vee \neg M_{i,j} \vee \neg B_j) \tag{1}$$
$$(\neg W_i \vee \neg M_{i,j} \vee B_j) \tag{2}$$

3.1.3 Driver Constraints. A wordline W_i can only carry an electrical current if there exists at least one bitline B_j that *drives* the wordline. A bitline B_j *drives* a wordline W_i if and only if the bitline B_j carries an electrical current, the memristor $M_{i,j}$ at their intersection is in a low resistive state, and the electrical current dissipates from the bitline to the wordline. Due to the bidirectionality of the memristors, the electrical current may also dissipate from the wordline to the bitline.

To define whether a wordline W_i is driven by any bitline B_j, we introduce driver variables $D_{i,j}$, $1 \leq i \leq N$ and $1 \leq j \leq M$, and a clause for each wordline:

$$(\neg W_i \vee D_{i,1} \vee D_{i,2} \vee \cdots \vee D_{i,N}) \tag{3}$$

Similarly, for each bitline B_j, we introduce a clause:

$$(\neg B_j \vee Q_{1,j} \vee Q_{2,j} \vee \cdots \vee Q_{M,j})$$

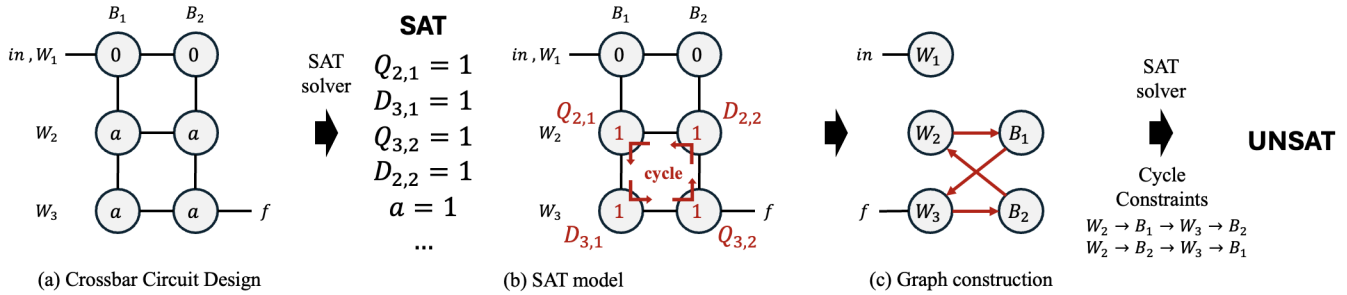
In total, there will be $2K$ driver variables if there are K memristors with a Boolean variable assigned. Memristor devices that are constantly on/off do not require driver variables.

3.1.4 Arrow Constraints. As discussed earlier, the driver variables $D_i, 1 \leq i \leq M$ determine whether a wordline is driven by a bitline or not. A bitline drives a wordline if and only if the bitline carries a current, the memristor at its intersection is in a low resistive state, and the electrical current flows from bitline to wordline. For the last constraint, a sense of directionality is required, which we introduce by means of an auxiliary *arrow* variable $A_{i,j}$. We will use the following definition for $A_{i,j}$:

$$A_{i,j} = \begin{cases} 1, & \text{if } B_j \rightarrow W_i \\ 0, & \text{if } W_i \rightarrow B_j \end{cases}$$

We add the following seven clauses:

$$(\neg D_{i,j} \vee M_{i,j} \vee A_{i,j} \vee W_i) \tag{4}$$
$$(\neg D_{i,j} \vee M_{i,j} \vee A_{i,j} \vee \neg W_i) \tag{5}$$
$$(\neg D_{i,j} \vee M_{i,j} \vee \neg A_{i,j} \vee W_i) \tag{6}$$
$$(\neg D_{i,j} \vee M_{i,j} \vee \neg A_{i,j} \vee \neg W_i) \tag{7}$$
$$(\neg D_{i,j} \vee \neg M_{i,j} \vee A_{i,j} \neg \vee W_i) \tag{8}$$
$$(\neg D_{i,j} \vee \neg M_{i,j} \vee \neg A_{i,j} \vee W_i) \tag{9}$$



(a) Crossbar Circuit Design (b) SAT model (c) Graph construction

Figure 4: Example of the cycle breaking. (a) Initially, we are given a crossbar circuit design that contains a cycle. (b) When the circuit is fed as part of its miter to the SAT solver, a model is found. A part of the model is shown, together with a graphical depiction where the drivers are indicated by red arrows. (c) Based on the model, a directed graph is constructed, from which the cycle can be deduced. Finally, the cycle is banned from the SAT formulation, and in subsequent iteration results in UNSAT.

3.2 Iterative Cycle Breaking

While the formulation in the previous section is able to handle cycles of length 2 using the arrow variables. The SAT formulation cannot handle cycles of length > 2. In this section, we propose a methodology to eliminate all such cycles automatically. A naive approach would be to enumerate all possible cycles in the undirected graph and ban them using sets of arrow variables. For example, a cycle containing the arrow variables A_1, A_2, A_3, A_4 can be banned by adding a clause $(\neg A_1 \vee \neg A_2 \vee \neg A_3 \vee \neg A_4)$. However, this is not practical because the number of possible cycles grows exponentially with the size of an undirected graph [12]. Instead, we analyze the solution from the SAT formulation to determine if the formulation became SAT due to a cycle or non-equivalence. If a cycle is detected, we explicitly ban that cycle using the arrow variables and resolve the SAT formulation. This process continues until the SAT formulation becomes UNSAT, or SAT, without any cycles. In Algorithm 1, we provide a high-level pseudocode for the overall iterative cycle-breaking algorithm. In subsequent sections, we will elaborate on the most important sub-steps: (1) graph construction, (2) cycle detection, and (3) cycle breaking.

We illustrate the proposed flow using the example in Figure 4. A 3×2 crossbar circuit is shown in Figure 4(a). The SAT solution and a cycle is show in Figure 4(b). The cycle is detected and banned in Figure 4(c). The SAT formulation results in UNSAT after the cycles have been banned.

Algorithm 1 Iterative SAT solving

Input: $\mathcal{D}, \mathcal{S}$ // Crossbar design and specification
Output: $\top$ or $\bot$ // True or false

```
1:  miter ←ConstructMiterCNF(D, S)
2:  model ←SolveSAT(miter) // Initialize set of cycles
3:  while model ≠ ∅ do
4:      graph ←ConstructGraph(model)
5:      cycles ←FindCycles(graph)
6:      if |cycles| = 0 then
7:          return ⊥
8:      end if
9:      miter←BreakCycles(miter, cycles)
10:     model←SolveSAT(miter)
11: end while
12: return ⊤
```

3.2.1 Graph Construction. Based on the found model, we construct a directed graph. This graph is a directed bipartite $B = (U_1, U_2, E)$ where U_1 is the set of wordlines of the crossbar, U_2 is the set of bitlines of the crossbar, and E is the set of edges. Each edge $e = (x, y)$ has one endpoint in U_1 and one in U_2. In a traditional graph model of the crossbar, the edges are undirected as the non-volatile memory devices are bidirectional. However, in graph B we will construct, the edges will be directed based on driver variables. In Algorithm 2, we provide the pseudo-code for the directed graph construction from the model. Note that either $D_{i,j}$ (wordline W_i drives bitline B_j) or $Q_{i,j}$ (bitline B_j drives wordline W_i) is true but not both at the same time.

Algorithm 2 Directed graph construction from model

```
1:  function ConstructGraph(model)
2:      graph ← ∅
3:      for i ∈ [1, M] do
4:          for j ∈ [1, N] do
5:              d ←GetValue(model, D_{i,j})
6:              q ←GetValue(model, Q_{i,j})
7:              if q then AddEdge(graph, i, j) end if
8:              if d then AddEdge(graph, j, i) end if
9:          end for
10:     end for
11:     return graph
12: end function
```

3.2.2 Cycle Detection. Next, we attempt to detect cycles within the unconnected component, if any. Since we have a directed graph, we leverage an adapted version of Johnson's algorithm [24]. For our work, we have leveraged an off-the-shelf implementation from NetworkX [15].

3.2.3 Cycle Constraints. Based on the cycles, we will now ban these from the model. We accomplish this by adding a new set of clauses and variables to the overall SAT formulation. Given a cycle indicated by the driver variables, we can ban the cycles. Say the cycle is supported by the drivers $Q_{1,0} \wedge D_{1,k} \wedge \cdots \wedge D_{l,0}$, then we can ban the cycle by disallowing the conjunction: $\neg(Q_{1,0} \wedge D_{1,k} \wedge \cdots \wedge D_{l,0})$. This conjunction is translated into CNF using the generic formula $z = \bigwedge_{i=1}^{j} x_i$ which becomes $(\bigwedge_{i=1}^{j}(x_i \vee \neg z))(\bigvee_{i=1}^{j} \neg x_i \vee z)$ [33]. Similar can be done for the reverse cycle.

4 Experiments

In this section, we evaluate our proposed FlowSAT framework. The framework is implemented in Python 3.10 and uses Glucose as SAT solver [4]. The code was executed on a Ubuntu machine with 20 i9 cores and 128GB RAM. We evaluate our framework on fourteen MCNC benchmarks [46] and six EPFL control benchmarks [3]. The source code is publicly available on GitHub [1]. In Table 1, we provide an overview of the properties of the benchmarks and the respective crossbar circuits. The crossbar circuit designs have been constructed using COMPACT, the state-of-the-art synthesis framework for flow-based computing [36]. Here, we have set the hyperparameter gamma to one, and we have used reduced ordered binary decision diagrams as data structure. For the crossbar circuits, the number of rows (wordlines), columns (bitlines), number of literals, and memristors programmed to true (ON) and false (OFF) are listed.

In subsequent sections, we will first compare our proposed FlowSAT framework with previous work [34, 40]. We will conduct a comparison both in the case where the crossbar circuit designs are both equivalent and non-equivalent to the specifications. Due to the nature of the crossbar circuit designs, our framework does not require iterations to solve the problem. Hence, we will conduct a sensitivity analysis of the FlowSAT framework in Section 4.2 on crossbar designs with cycles.

4.1 Comparison with Previous Work

We compare our framework with three other equivalence checking techniques for flow-based computing. These techniques are brute-force enumeration [10], recurrent neural networks (RNNs) [34], and the state-of-the-art equivalence checking framework. In subsequent sections, we compare these techniques for both equivalent and non-equivalent crossbar circuit designs. For both cases, we report the most important results for each of the techniques. For enumeration and the RNN-based technique, we report the total runtime (in seconds). For the previous state-of-the-art method XSAT, we report the number of time steps for which the circuit is unrolled/duplicated for bounded model checking (steps), the number of variables and clauses for the SAT formulation, and a breakdown of the runtime. For the runtime, we distinguish between the time to perform pre-processing (Pre) and the time to perform combinational equivalence checking (CEC). Finally, we also report the total runtime in seconds. The XSAT framework makes use of the ABC tool [6] to perform combinational equivalence checking, which is in turn built upon MiniSAT [35]. For our proposed FlowSAT framework, we also report the same numbers as XSAT, except for the number of time steps the circuit is unrolled, as this is not applicable to our approach. Note that the total runtime may slightly differ from the pre-processing time plus the time for combinational equivalence checking as there is some overhead in other parts of the code.

For the experiments, we set a timeout of one hour (3600 seconds). When the equivalence checking technique times out, we indicate this by a '-' for the results.

4.1.1 Case 1: Equivalence. In our first experiment, we will evaluate the frameworks on crossbar circuit designs that are equivalent to their specifications. In Table 2, an overview is provided of the results for all four frameworks. First, we observe that brute-force enumeration does not perform well on these large crossbar circuit designs as the number of input variables is high. For smaller designs, such as *cavlc*, we can only evaluate up to eleven input variables. We conclude that brute-force enumeration is only capable of proving equivalence for three out of twenty designs within one hour. Next, for the RNN-based approach, we observe that we can successfully show equivalence for twelve out of twenty designs. The technique is fast for benchmarks with relatively small number of input variables (less than thirty). For example, *misex3* and *bc0* have 14 and 21 input variables, respectively, and are consequently evaluated in less than two minutes. Other benchmarks, such as *x3* and *apex3* timeout as both have 135 and 54 input variables, respectively. In summary, the RNN approach is capable of showing equivalence for twelve out of twenty benchmarks. For XSAT, the first observation is that the number of time steps for which the crossbar circuit is unrolled is high (hundreds to thousands of times). This results in a high number of variables and clauses ranging from millions to hundred millions. The discrepancy in the orders of magnitude for XSAT and FlowSAT stems from the duplication of the circuit in bounded model checking. The number of variables for XSAT grows with $O((M+N)\cdot(NM)+K)$ where M is the number of wordlines, N the number of bitlines, and K the number of non-zero literals in the crossbar. The number of variables for FlowSAT grows with $O(M+N+K)$ on the other hand. We observe that XSAT can only show equivalence for twelve out of twenty benchmarks within one hour. Further, we also observe that the CEC time is not strongly correlated with the number of clauses and variables. For example, benchmark *bc0* has almost four million clauses whereas *b2* has about 26 million clauses, yet the latter was completed earlier than the former. Finally, we analyze

Table 1: Overview of 20 benchmarks (14 MCNC and 6 EPFL control). For each benchmark, we report the number of inputs and outputs. For the corresponding crossbar design, we report the number of rows, columns, literals, and the number of memristors that are programmed ON and OFF.

Benchmark	Properties		Crossbar				
	Inputs (num)	Outputs (num)	Rows (num)	Columns (num)	Literals (num)	ON (num)	OFF (num)
MCNC							
in4	32	20	432	411	1209	89	176254
too_large	38	3	439	433	1411	119	188557
misex3	14	14	486	495	1403	82	239085
bc0	21	11	517	522	1571	86	268217
pdc	16	40	604	629	1578	61	378277
alu4	14	8	605	632	2159	114	380087
spla	16	46	626	579	1513	66	360875
vda	17	39	680	666	1901	112	450867
x3	135	99	700	576	1772	124	401304
apex6	135	99	700	576	1772	124	401304
apex3	54	50	772	802	2262	56	616826
ex1010	10	10	850	812	2695	60	687445
b2	15	17	1001	982	2839	139	980004
prom1	9	40	1980	1765	6246	160	3488294
EPFL							
cavlc	10	11	317	303	989	39	95023
dec	8	256	1025	1024	2048	0	1047552
i2c	147	142	1345	1279	3492	210	1716553
int2float	11	7	122	121	394	29	14339
priority	128	8	504	520	1786	126	260168
router	60	30	125	122	390	22	14838

[1] https://github.com/sventhijssen/flowsat

Table 2: Comparison for twenty benchmarks where the crossbar circuit design is equivalent to the specification. The benchmarks are evaluated on four equivalence checking techniques for flow-based computing: brute-force enumeration, a RNN-based approach, the XSAT framework, and the proposed FlowSAT framework. For timeouts ('-'), a cost of $3600s$ is calculated for the normalized runtime comparison.

Benchmark	Enum [10]	RNN [34]	XSAT [40]						FlowSAT				
	Total (s)	Total (s)	Steps (num)	Variables (num)	Clauses (num)	Pre (s)	CEC (s)	Total (s)	Variables (num)	Clauses (num)	Pre (s)	CEC (s)	Total (s)
MCNC													
in4	-	-	754	5879525	2930689	21.0	1248.7	1272.7	533913	1433718	30.6	2.4	33.0
too_large	-	-	753	7467553	3713647	25.5	1396.3	1424.8	572056	1548459	34.0	5.4	39.5
misex3	-	0.845	899	7425919	3705516	26.5	2236.0	2265.5	723265	1939814	39.9	3.4	43.4
bc0	-	71.55	953	7991101	3988632	31.7	2294.5	2329.3	811185	2175622	43.1	3.8	46.9
pdc	-	3.158	-	-	-	-	-	-	1141533	3058378	62.9	6.9	69.8
alu4	-	1.355	-	-	-	-	-	-	1148826	3078253	64.6	8.7	73.3
spla	-	3.245	-	-	-	-	-	-	1089118	2917542	57.0	6.2	63.2
vda	-	8.826	-	-	-	-	-	-	1360304	3640531	71.4	7.8	79.2
x3	-	-	1152	8106624	4047922	42.5	1054.5	1100.1	1212089	3242526	64.2	6.8	71.0
apex6	-	-	1152	8452357	4220578	42.4	1107.5	1152.9	1211775	3241422	66.7	7.3	74.0
apex3	-	-	-	-	-	-	-	-	1859772	4976558	97.2	91.2	188.5
ex1010	-	0.558	1602	66579644	23596048	88.4	1095.7	1187.1	2073258	5553986	109.4	147.1	256.4
b2	-	6.509	1844	74341323	26188553	104.3	1779.7	1887.1	2951704	7893154	154.4	174.8	329.2
prom1	-	1.177	-	-	-	-	-	-	10489973	28026661	567.3	1692.6	2260.0
EPFL													
cavlc	893.0	0.5	581	8482411	2992814	14.2	83.0	100.2	290180	779684	16.3	0.8	17.1
dec	2749.6	0.5	2049	32834685	11467647	92.0	76.2	171.3	3151721	8417521	164.1	138.9	303.0
i2c	-	-	-	-	-	-	-	-	5166391	13796612	272.9	482.0	754.9
int2float	278.7	0.5	214	1166970	408620	5.7	7.3	16.1	45067	122491	3.2	0.1	3.4
priority	-	-	-	-	-	-	-	-	789356	2115476	43.2	3.8	46.9
router	-	-	225	589049	282800	5.7	39.6	48.3	46628	126094	3.3	0.1	3.4
Total	3/20	12/20						12/20					**20/20**
Normalized	1.00	0.40						0.59					**0.07**

our proposed FlowSAT framework and compare it with XSAT. First, we observe that the number of variables and clauses for FlowSAT is much smaller than for XSAT. On average, FlowSAT only requires 8% and 49% of the number of variables and clauses, respectively, compared with XSAT. Consequently, this results in much faster runtimes for the proposed SAT formulation. In summary, FlowSAT is the only framework capable of showing equivalence between the crossbar circuit design and the specifications within one hour for all twenty benchmarks.

4.1.2 Case 2: Non-Equivalence. In this section, we evaluate the four frameworks for non-equivalent crossbar circuit designs. To construct these designs, we replace one literal at a random position in the crossbar design with either a logical zero or one. In Table 3, we report the results for the same properties as in the previous section. Our first observation is that all approaches succeed for at least as many benchmarks as in the equivalent case. This is because it suffices to find at least one counter-example for which the design and the specification are not equivalent. For brute-force enumeration, twelve out of twenty designs are completed within one hour compared with three. For the RNN-based approach, the results are the same as the framework relies on the construction of a full truth table, which requires the same amount of time as with the equivalent case. The XSAT framework is now capable of completing all benchmarks within one hour. We observe that the runtime has decreased significantly for many benchmarks. For example, where *pdc* was not able to complete in one hour in the equivalent case, non-equivalence can now be shown in about 80 seconds. This is due to the number of variables and clauses being smaller than in the equivalent case as a consequence of the pre-processing within the ABC tool. For example, for *cavlc*, non-equivalence can be shown in the pre-processing stage, where and-inverter graphs are constructed before the miter is even fed to the SAT solver. Lastly, the runtime results for FlowSAT are somewhat similar to those of the equivalent case, with some runtimes being slightly shorter or longer. As we can observe, the number of variables and clauses are similar to numbers in the equivalent case; hence, the runtimes are similar, only slightly influenced by external factors. On average, the overall runtime for our proposed FlowSAT framework is 20% faster than XSAT, capable of completing all twenty benchmarks within one hour.

4.2 Sensitivity Analysis

Finally, we perform a sensitivity analysis of our proposed FlowSAT framework on designs with many cycles being added iteratively and incrementally. In previous experiments, the benchmarks were based on binary decision diagrams (BDDs). Due to inherent characteristics of BDDs (Shannon's expansion), cycles of length 2 mainly occur. Hence, in this experiment, we start with a random crossbar design of dimensions 32×32 with 5% density using 10 input variables. Then we add additional literals at random positions within the design, which will eventually add cycles to the design. In Figure 5(a) and (b), we show the number of variables and clauses, respectively, for an

Table 3: Comparison for twenty benchmarks where the crossbar circuit design is non-equivalent to the specification. The benchmarks are evaluated on four equivalence checking techniques for flow-based computing: brute-force enumeration, a RNN-based approach, the XSAT framework, and the proposed FlowSAT framework.

Benchmark	Enum [10]	RNN [34]	XSAT [40]						FlowSAT				
	Total (s)	Total (s)	Steps (num)	Variables (num)	Clauses (num)	Pre (s)	CEC (s)	Total (s)	Variables (num)	Clauses (num)	Pre (s)	CEC (s)	Total (s)
MCNC													
in4	-	-	753	14337210	5063400	20.9	23.9	47.8	533913	1433718	30.0	2.0	32.0
too_large	-	-	752	16959636	6057377	26.0	328.7	357.8	572056	1548459	31.6	2.3	33.8
misex3	1147.1	0.8	898	19211621	6754731	27.7	60.9	91.5	723265	1939814	39.1	3.1	42.2
bc0	-	71.55	952	20796575	7340488	32.2	22.6	57.8	811185	2175622	44.2	3.7	47.9
pdc	813.4	3.2	1171	28226342	9813103	42.2	35.8	80.9	1141533	3058378	61.8	5.8	67.6
alu4	12.1	1.4	1122	36374323	12955187	49.7	101.9	154.5	1148826	3078253	62.2	6.1	68.2
spla	671.5	3.2	1138	26070468	9044608	38.7	36.4	78.1	1089118	2917542	58.4	5.6	64.0
vda	321.1	8.8	1233	34896576	12157489	48.0	34.7	85.7	1360304	3640531	73.5	7.7	81.2
x3	-	-	1151	26912747	9537253	41.1	39.8	83.9	1212089	3242526	66.1	6.6	72.7
apex6	-	-	1151	26980455	9572869	40.3	152.1	195.4	1211775	3241422	66.1	6.7	72.8
apex3	-	-	1517	45292867	15785619	68.6	1384.4	1456.1	1859772	4976558	100.3	88.7	189.0
ex1010	638.1	0.6	1601	66541208	23583274	85.0	280.5	368.5	2073258	5553986	111.5	152.8	264.3
b2	-	6.509	1843	74285477	26168954	106.1	142.0	251.1	2951704	7893154	162.4	163.2	325.6
prom1	391.4	1.2	3584	322233451	113768411	429.9	796.8	1229.7	10489973	28026661	565.2	1490.0	2055.2
EPFL													
cavlc	0.9	0.5	580	0	0	14.1	4.9	22.0	290180	779684	16.6	0.8	17.4
dec	801.6	0.5	2048	33034600	11542261	90.1	42.0	135.1	3151721	8417521	172.6	143.6	316.2
i2c	298.4	-	2413	120477837	41897111	171.9	394.2	569.2	5166391	13796612	284.0	440.2	724.2
int2float	0.1	0.5	213	0	0	5.8	0.5	9.3	45067	122491	3.3	0.1	3.4
priority	8.3	-	897	22502866	7991622	31.9	37.1	72.1	789356	2115476	43.9	3.8	47.7
router	-	-	224	9233	3660	6.0	16.0	24.9	46628	126094	5.6	0.1	5.7
Total	12/20	12/20						**20/20**					**20/20**
Normalized	1.0	22.7						5.8					3.1

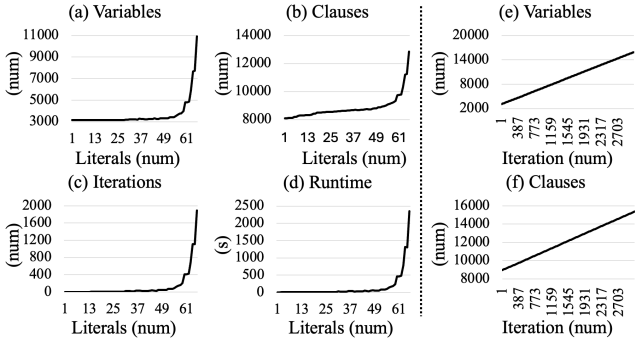


Figure 5: Sensitivity analysis on a 32×32 random crossbar design with 10 input variables. In (a), (b), (c), and (d), the number of variables, clauses, iterations, and the runtime in seconds is shown for our proposed FlowSAT framework for increasingly dense designs. In (e) and (f), the number of variables and clauses are shown for a single run that requires multiple iterations for cycle breaking.

increasing number of literals (without the variables and clauses of the specification as this may shrink). We observe that the number grows rapidly from some point. This is in accordance with the theoretical results in [44] that there is potentially a factorial number of paths within a crossbar. Further, the number of iterations, and consequently the runtime grows accordingly, as shown in Figure 5(c) and (d). In Figure 5(e) and (f), we show the growth of the number of variables and clauses for a single run (for 66 added literals), and we observe a linear growth in both magnitudes in terms of the number of iterations. This highlights the scalability of the proposed approach even in the presence of many cycles.

5 Conclusion

Flow-based computing is an energy-efficient in-memory computing paradigm on crossbar arrays. Due to the bidirectional behavior of non-volatile memory devices, the crossbar circuit is modeled as an undirected graphs, contrasting with traditional VLSI circuits, which are modeled as directed acyclic graphs. This has prevented the use of SAT formulations for equivalence checking. Hence, for verification, alternative methods have been developed based on brute-force enumeration, graph reachability, recurrent neural networks, and bounded model checking. Unfortunately, these methods do not scale well when the number of input variables is large and/or when the dimensions of the crossbar circuit designs are large. In this paper, we have introduced a SAT formulation that can cope with the non-traditional circuit behavior. We have solved the problem of bidirectionality by introducing auxiliary variables and arrow variables, which provide a sense of directionality. Further, to cope with the possible cyclic behavior of the circuit, we rely on an iterative algorithm to detect such cycles and remove them from the SAT formulation in the next iteration. From our experimental evaluation, we conclude that our proposed FlowSAT framework is up to an order of magnitude faster than the previous state-of-the-art method. For future work, we suggest the exploration of model checking [11], automated synthesis [20], and AI-based methods [22].

References

[1] [n. d.]. Joint University Microelectronics Program. https://www.darpa.mil/program/joint-university-microelectronics-program.

[2] T Ali, P Polakowski, S Riedel, T Büttner, T Kämpfe, M Rudolph, B Pätzold, K Seidel, D Löhr, R Hoffmann, et al. 2018. High endurance ferroelectric hafnium oxide-based FeFET memory without retention penalty. *IEEE Transactions on Electron Devices* 65, 9 (2018), 3769–3774.

[3] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*.

[4] Gilles Audemard and Laurent Simon. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence* (Pasadena, California, USA) *(IJCAI'09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 399–404.

[5] Julien Borghetti, Gregory S Snider, Philip J Kuekes, J Joshua Yang, Duncan R Stewart, and R Stanley Williams. 2010. 'Memristive'switches enable 'stateful'logic operations via material implication. *Nature* 464, 7290 (2010), 873–876.

[6] Robert K. Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[7] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. 2010. Phase change memory technology. *Journal of Vacuum Science & Technology B* 28, 2 (2010), 223–262.

[8] Dwaipayan Chakraborty and Sumit Kumar Jha. 2017. Automated synthesis of compact crossbars for sneak-path based in-memory computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 770–775.

[9] Dwaipayan Chakraborty and Sumit Kumar Jha. 2017. Design of compact memristive in-memory computing systems using model counting. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

[10] Dwaipayan Chakraborty and Sumit Kumar Jha. 2017. Design of compact memristive in-memory computing systems using model counting. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

[11] Edmund Clarke, Ansgar Fehnker, Sumit Kumar Jha, and Helmut Veith. 2005. Temporal logic model checking. *Handbook of Networked and Embedded Control Systems* (2005), 539–558.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[13] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*. 365–376.

[14] Pierre-Emmanuel Gaillardon, Luca Amarú, Anne Siemon, Eike Linn, Rainer Waser, Anupam Chattopadhyay, and Giovanni De Micheli. 2016. The programmable logic-in-memory (PLiM) computer. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Ieee, 427–432.

[15] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[16] Amad Ul Hassen, Dwaipayan Chakraborty, and Sumit Kumar Jha. 2018. Free binary decision diagram-based synthesis of compact crossbars for in-memory computing. *IEEE Transactions on Circuits and Systems II: Express Briefs* 65, 5 (2018), 622–626.

[17] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R Stanley Williams. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Proceedings of the 53rd annual design automation conference*. 1–6.

[18] Yiming Huai et al. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin* 18, 6 (2008), 33–40.

[19] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*. 802–815.

[20] Susmit Kumar Jha. 2011. *Towards automated system synthesis using sciduction*. University of California, Berkeley.

[21] Sumit Kumar Jha, Susmit Jha, Rickard Ewetz, and Alvaro Velasquez. 2024. On the Design of Novel Attention Mechanism for Enhanced Efficiency of Transformers. In *61st ACM Design Automation Conference (DAC)*.

[22] Sumit Kumar Jha, Susmit Jha, Muhammad Rashedul Haq Rashed, Rickard Ewetz, and Alvaro Velasquez. 2024. Automated Synthesis of Hardware Designs using Symbolic Feedback and Grammar-Constrained Decoding in Large Language Models. In *NAECON 2024 - IEEE National Aerospace and Electronics Conference*.

[23] Sumit Kumar Jha, Dilia E Rodriguez, Joseph E Van Nostrand, and Alvaro Velasquez. 2016. Computation of boolean formulas using sneak paths in crossbar computing. US Patent 9,319,047.

[24] Donald B Johnson. 1975. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4, 1 (1975), 77–84.

[25] Florian Krohm, Andreas Kuchlmann, and Arjen Mets. 1996. The use of random simulation in formal verification. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*. IEEE, 371–376.

[26] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. 2014. MAGIC—Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[27] Shahar Kvatinsky, Misbah Ramadan, Eby G Friedman, and Avinoam Kolodny. 2015. VTEAM: A general model for voltage-controlled memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 8 (2015), 786–790.

[28] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[29] Paul Messina. 2017. The exascale computing project. *Computing in Science & Engineering* 19, 3 (2017), 63–67.

[30] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. 2006. Improvements to combinational equivalence checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. 836–843.

[31] Jodh Singh Pannu, Sunny Raj, Steven Lawrence Fernandes, Dwaipayan Chakraborty, Sarah Rafiq, Nathaniel Cady, and Sumit Kumar Jha. 2020. Design and fabrication of flow-based edge detection memristor crossbar circuits. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 5 (2020), 961–965.

[32] Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2021. Hybrid analog-digital in-memory computing. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[33] Jarrod A Roy, Igor L Markov, and Valeria Bertacco. 2004. Restoring circuit structure from SAT instances. *Ann Arbor* 1001 (2004), 48109–2122.

[34] Suraj Singireddy, Rickard Ewetz, and Sumit Jha. 2022. Deep learning toolkit-driven equivalence checking of flow-based computing systems. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 50–53.

[35] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.

[36] Sven Thijssen, Sumit Kumar Jha, and Rickard Ewetz. 2021. Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter and maximum dimension. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2021), 4600–4611.

[37] Sven Thijssen, Sumit Kumar Jha, and Rickard Ewetz. 2022. Equivalence checking for flow-based computing. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 656–663.

[38] Sven Thijssen, Sumit Kumar Jha, and Rickard Ewetz. 2022. Path: Evaluation of boolean logic using path-based in-memory computing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1129–1134.

[39] S. Thijssen, Muhammad Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2024. Synthesis of Compact Flow-based Computing Circuits from Boolean Expressions. In *61st ACM Design Automation Conference (DAC)*.

[40] Sven Thijssen, Suraj Singireddy, Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2023. Verification of Flow-Based Computing Systems Using Bounded Model Checking. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[41] Elie Track, Nancy Forbes, and George Strawn. 2017. The end of Moore's Law. *Computing in Science & Engineering* 19, 2 (2017), 4–6.

[42] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* (1983), 466–483.

[43] Alvaro Velasquez and Sumit Kumar Jha. 2014. Parallel computing using memristive crossbar networks: Nullifying the processor-memory bottleneck. In *2014 9th International Design and Test Symposium (IDT)*. IEEE, 147–152.

[44] Alvaro Velasquez and Sumit Kumar Jha. 2015. Fault-tolerant in-memory crossbar computing using quantified constraint solving. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 101–108.

[45] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[46] Saeyang Yang. 1991. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer.

[47] Shimeng Yu. 2016. Resistive random access memory (RRAM). *Synthesis Lectures on Emerging Engineering Technologies* 2, 5 (2016), 1–79.

[48] Yue Zha and Jing Li. 2016. Reconfigurable in-memory computing with resistive memory crossbar. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2967069