Synthesis of Compact Flow-based Computing Circuits from Boolean Expressions

Sven Thijssen
sven.thijssen@ucf.edu
University of Central Florida
Orlando, FL, USA

Muhammed Rashedul Haq Rashed
muhammad.rashed@ucf.edu
University of Central Florida
Orlando, FL, USA

Sumit Kumar Jha
sumit.jha@fiu.edu
Florida International University
Miami, FL, USA

Rickard Ewetz
rickard.ewetz@ucf.edu
University of Central Florida
Orlando, FL, USA

ABSTRACT

Processing in-memory has the potential to accelerate high-data-rate applications beyond the limits of modern hardware. Flow-based computing is a computing paradigm for executing Boolean logic within nanoscale memory arrays by leveraging the natural flow of electric current. Previous approaches of mapping Boolean logic onto flow-based computing circuits have been constrained by their reliance on binary decision diagrams (BDDs), which translates into high area overhead. In this paper, we introduce a novel framework called FACTOR for mapping logic functions into dense flow-based computing circuits. The proposed methodology introduces Boolean connectivity graphs (BCGs) as a more versatile representation, capable of producing smaller crossbar circuits. The framework constructs concise BCGs using factorization and expression trees. Next, the BCGs are modified to be amenable for mapping to crossbar hardware. We also propose a time multiplexing strategy for sharing hardware between different Boolean functions. Compared with the state-of-the-art approach, the experimental evaluation using 14 circuits demonstrates that FACTOR reduces area, speed, and energy with 80%, 2%, and 12%, respectively, compared with the state-of-the-art synthesis method for flow-based computing.

ACM Reference Format:
Sven Thijssen, Muhammed Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2024. Synthesis of Compact Flow-based Computing Circuits from Boolean Expressions. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657340

1 INTRODUCTION

Advances in computer hardware and system have long relied on Moore's law for continuous performance improvements [10]. However, we are quickly approaching the physical limit of the minimum reliable transistor size. Moreover, today's computing systems are also constrained by the von Neumann bottleneck [15] which separates a system into two key components: the central processing unit (CPU) and the memory unit(s). The majority of data is stored in the memory unit and is shuttled to and from the CPU when a computation takes place. The limited bandwidth of the data bus between the two units has a negative effect on the overall computing performance for high-data-rate applications. This has spurred the interest in emerging computing technologies and paradigms for future computing systems. Recent investigations include quantum computing [2], photonic computing [14], and in-memory computing using non-volatile memory (NVM) [12].

Many in-memory computing paradigms using memristors have been proposed, such as Memristor Ratioed Logic (MRL) [7], Memristor-Based Material Implication (IMPLY) [8], Memristor-Aided Logic (MAGIC) [9], and flow-based computing [1]. However, MRL operates on voltage levels instead of stored memory, IMPLY suffers from being destructive, and MAGIC is mainly efficient for parallel and regular computational patterns [11]. In contrasts, flow-based computing has been shown to be efficient for executing irregular Boolean functions.

Flow-based computing is based on programming the non-volatile memory devices within a nanoscale crossbar to a specific pattern based on an input instance of a Boolean function. This layout is designed such that two wires within the crossbar will be connected by a low-resistance path if and only if the data stored in the memristors satisfy a desired logic function. This allows for near-instant execution of digital functions by applying a voltage to the topmost wordline and measuring the output current/voltage from the bottom-most wordline. While the design of these crossbar circuits can be done by hand [1], automated methods have recently been explored in [4]. A crucial design choice within these methods is the selection of the representation for the Boolean function. Techniques have been developed based on disjunctive/conjunctive normal form (DNF/CNF) [17], binary decision diagrams (BDDs) [5], and free binary decision diagrams (FBDDs) [6]. The state-of-the-art tool COMPACT [16] is based on first converting Boolean functions into BDDs, which are directly mapped into crossbar designs for flow-based computing. With respect to the provided BDD, COMPACT is optimal in terms of hardware resources. Unfortunately, the size of BDDs are known to scale poorly for some Boolean functions, which translates into high area overheads.

The authors acknowledge support from NSF awards #2319399, #2408925, and #2404036.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657340

In this paper, we introduce FACTOR, a framework for mapping Boolean functions to area-efficient flow-based computing circuits. The framework is based on introducing Boolean connectivity graphs (BCGs), which are used to concisely represent Boolean functions. The BCGs allow a single node to have more than two outgoing edges, which leads to an inherent advantage over BDD-based approaches. Next, the BCGs are slightly modified to satisfy crossbar imposed hardware constraints, and the resulting graph is subsequently bound to a nanoscale array. While BCGs are straightforward for single-output Boolean functions, the construction of BCGs for multi-output Boolean functions is achieved using time multiplexing. The time multiplexing allows hardware reuse across different outputs, which translates to further area savings. We evaluate our proposed FACTOR framework on 14 Revlib benchmarks. Compared with the previous state-of-the-art synthesis method for flow-based computing [16], FACTOR reduces area, latency, and energy by 80%, 2%, and 12% on average.

In Section 2, we provide preliminaries on negation normal form, crossbar arrays, and flow-based computing. In Section 3, we provide motivation for our framework and in Section 4, we introduce the FACTOR framework. An extension to sharing logic in multi-output Boolean functions is provided in Section 5. Experimental evaluation is discussed in Section 6, and the paper is concluded in Section 7.

2 PRELIMINARIES

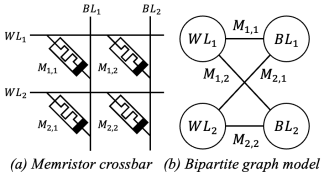
2.1 Negation Normal Form and Factoring

A Boolean expression in Negation Normal Form (NNF) is characterized by the ¬ (not) operation being exclusively applied to individual Boolean variables, and otherwise only uses the Boolean operators ∧ (logical AND) and ∨ (logical OR). Any arbitrary Boolean expression can be converted into NNF form by applying De Morgan's law to any ¬ operation that is applied to more than a single literal. The simplification may need to be applied recursively to the subexpressions produced, until the expression is in NNF form. The NNF form is of interest because crossbars within flow-based computing only supports negation operations on the primary inputs.

Boolean functions in NNF form can be factored to reduce the literal count. Factoring involves extracting subexpressions that are common for multiple parts of a function or multi-output functions. For example, the Boolean function $f = (a \wedge c) \vee (\neg b \wedge c)$ can be factored into $f = (a \vee \neg b) \wedge c$, which reduces the literal count from four to three. Several logic synthesis tools are available for this purpose, including SIS [13].

2.2 Memristor Crossbars and Flow-based Computing

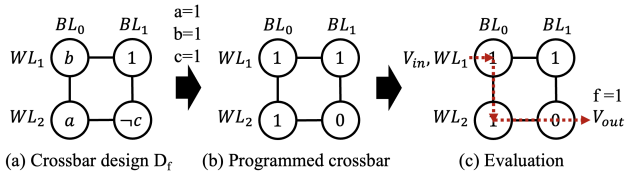
Nanoscale crossbar arrays organize memristors into a cross-point structure using wordlines and bitlines to store large amounts of data [12]. A memristor is placed at each intersection, with one end connected to the nearby wordline and the other to the nearby bitline.



(a) Memristor crossbar (b) Bipartite graph model

Figure 1: Memristor crossbar and bipartite graph model.

An example of a memristor crossbar is shown in Figure 1(a). A memristor crossbar can be modeled as a bipartite graph B where each wordline WL_i and bitline BL_j are modeled as a node. The memristor M_{ij} at their intersection is modeled as an edge between the nodes WL_i and BL_j. The graph model is shown in Figure 1(b).



(a) Crossbar design D_f (b) Programmed crossbar (c) Evaluation

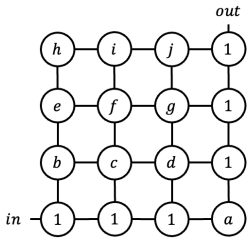
Figure 2: Example of flow-based Computing

Flow-based computing is a digital in-memory computing paradigm, executing Boolean functions using a 3-stage process on nanoscale crossbars. First, an abstract crossbar design is synthesized from a Boolean function. The design dictates where Boolean variables (or their complements) should be stored in the memristor array. A crossbar representation of the Boolean function $f = (a \wedge b) \vee \neg c$ is shown in Figure 2(a). Second, the crossbar hardware is programmed with respect to an instance of the Boolean variables. In Figure 2(b), the crossbar is programmed with respect to the instance of the Boolean variables $(a, b, c) = (1, 1, 1)$. Third, the function is evaluated by applying a positive voltage to the designated *in* wordline and measuring the voltage at the designated *out* wordline alongside a pull-down resistor. The voltage will only be high if there exists a low-resistance path between the *in* and *out* wordlines through the memristors with a stored value of 1. The function is shown to evaluate to *true* in Figure 2(c).

3 MOTIVATION

3.1 Limitations of BDD-based synthesis

There is an inherent limitation of synthesizing crossbars from BDDs. Each node in a BDD has at most 2 output edges, so the total number of edges can be no more than twice the number of nodes. On a crossbar, the nodes represent rows and columns, and the edges represent used memristors (which are not permanently set to 0 or 1). Thus—denoting the number of used memristors n—the number of rows and columns will be $\in \Theta(n)$, and the number of intersections in the crossbar will be $\in \Theta(n^2)$. For large n, this creates crossbars that are very sparsely populated by used memristors, which is not efficient in terms of area.



Area cost for N-input OR

Inputs	BDD	Optimal
1	1	1
10	100	16
100	10,000	121
10000	1,000,000	1,089

Figure 3: (a) Crossbar design for a 10-input OR, and (b) an overview of the area cost for an N-input OR using either BDDs or their optimal solutions for varying values of N.

To justify our claim, we perform a small case study in Figure 3. We illustrate an optimal crossbar design for a 10-input OR function

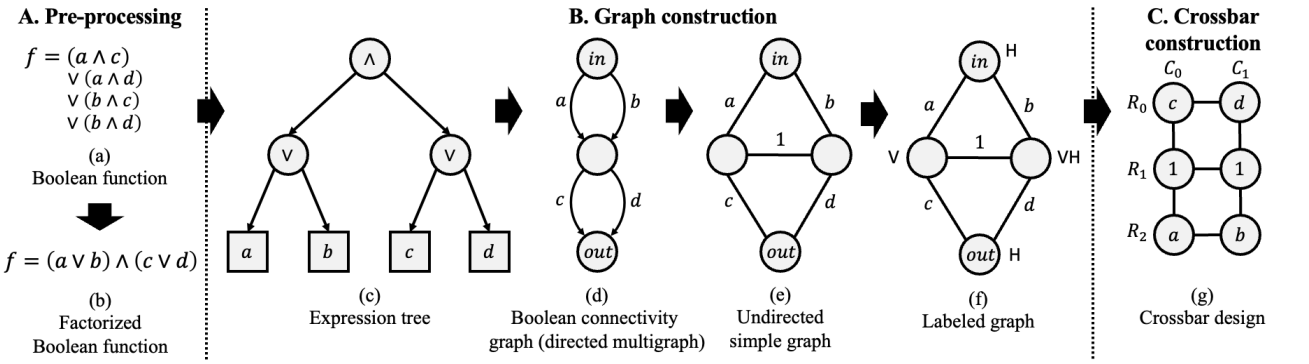


Figure 6: An example of the synthesis flow using the FACTOR framework. (a) The input is a Boolean function, which is subsequently factorized (b). Next, the factorized expression is converted into an expression tree in (c). From this tree, a Boolean connectivity graph is constructed in (d), which is a directed multigraph. Subsequently, the directed multigraph is converted into an undirected simple graph in (e), which is then labeled in (f). Finally, a crossbar design is constructed in (g).

in a crossbar with dimensions 4×4 in Figure 3(a). The equivalent layout produced by a BDD can be no smaller than 10×10. Clearly, using an alternative data structure, it is possible to construct crossbar designs which are more dense, and consequently have smaller area. In the next section, we introduce Boolean connectivity graphs, which can be used to synthesize such dense crossbar designs.

3.2 Proposed Boolean Connectivity Graphs (BCGs)

In this paper, we propose the use of Boolean Connectivity Graphs (BCGs). A BCG is a graph representation of a Boolean function f. The graph $G = (V, E)$ has a set of nodes V and a set of edges E labeled with Boolean literals $\{x_0, \neg x_0, ..., x_n, \neg x_n\}$ or Boolean truth values $\{0, 1\}$.

In contrast with BDDs, the graph is a directed multigraph where each node may have more than two outgoing edges. Binary decision diagrams (BDDs) are a subset of BCGs and are commonly used to synthesize memristor crossbars. They have certain structural restrictions that, while making them easier to map to layouts, prevent them from accessing the full solution space. In Figure 4(a), we show a BDD and in Figure 4(b), we show an equivalent BCG. The BDD has five nodes (including terminal nodes) and six edges whereas the BCG has three nodes and three edges. In our proposed synthesis method, we will leverage this alternative data structure to construct more succinct crossbar designs than with BDDs.

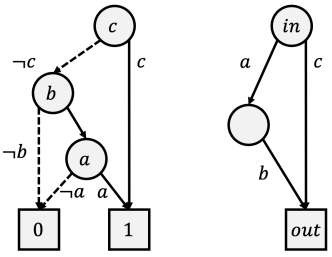


Figure 4: Comparison of graph size for a BDD and a BCG representing the same Boolean function $f = (a \wedge b) \vee c$.

4 THE FACTOR FRAMEWORK

In this section, we introduce our proposed FACTOR framework. The input to FACTOR is a single-output Boolean function f and the output is a crossbar design D_f realizing the Boolean function f. The framework consists of three main steps: (1) pre-processing, (2) graph construction, and (3) crossbar construction. A high-level overview of the FACTOR framework is shown in Figure 5. For the crossbar construction, we seek a graph representation for a one-to-one mapping between the graph and the crossbar. Based on the graph properties of a crossbar design outlined in Section 2.2, we identify two properties the graph must satisfy:

I. **Property I:** The graph must be a simple graph. This entails that for every pair of nodes, there must be at most one edge between them.
II. **Property II:** The graph must be a bipartite graph.

Another main objective of the FACTOR framework is to identify multi-input OR operations, which can be realized efficiently using the dense crossbar structure, as shown in Figure 3(a).

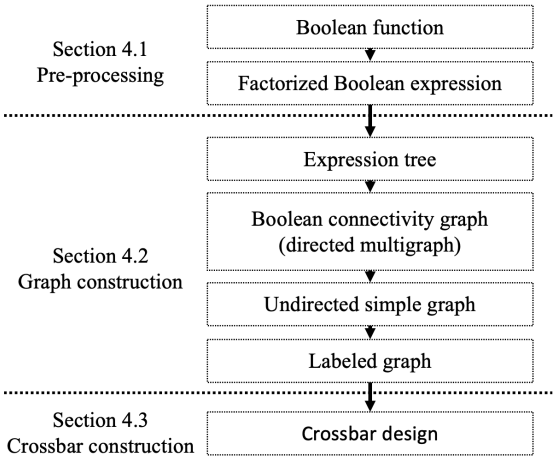


Figure 5: Overview of the FACTOR framework.

4.1 Pre-processing

The input of the first step is the Boolean function f, as shown in Figure 6(a). The Boolean function f is described in a hardware descriptive language such as Verilog or VHDL. Next, the Boolean function f is converted into NNF and factorized, as shown in Figure 6(b). For factorization, we use the logic synthesis tool SIS [13].

4.2 Graph construction

In this section, we construct a labeled graph from the given factorized Boolean function f using a Boolean connectivity graph. The graph construction has four substeps: (1) construction of an expression tree, (2) construction of a Boolean connectivity graph, (3), construction of an undirected simple graph, and (4) construction of a labeled graph.

4.2.1 Expression tree. First, we construct an expression tree from the factorized Boolean function, as shown in Figure 6(c). All nodes in the tree, except for the leaf nodes, are labeled with the Boolean operators $\wedge$ or $\vee$. The leaf nodes are labeled with Boolean literals $\{x_1, \neg x_1, ..., x_n, \neg x_n\}$. The expression tree is constructed bottom-up where the binary operators are converted into nodes according to the order of operations. These nodes have two outgoing edges, one to each operand.

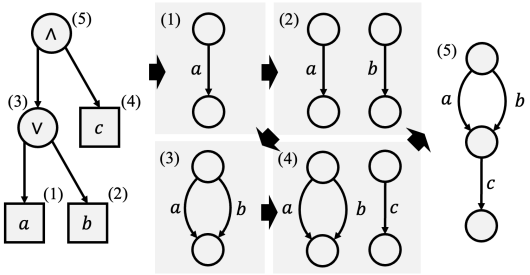


Figure 7: Construction of a Boolean connectivity graph (a directed multigraph) from an expression tree using a recursive algorithm. Each node corresponds to a recursive step and is indicated by a number between brackets, from which a BCG is subsequently constructed.

4.2.2 Boolean connectivity graph. Given the expression tree, we construct a Boolean connectivity graph using the recursive Algorithm 1. The Boolean connectivity graph is a directed multigraph where the edges are labeled with literals. The algorithm works as follows: first, it constructs the graphs for the left and right sub-trees of the expression. Then, it merges those graphs based on the top-level operation for the expression. If the operation is a conjunction ($\wedge$), then the two graphs are placed into series. Otherwise, if the operation is a disjunction ($\vee$), then the two graphs are placed in parallel. It is precisely this disjunction operation which provides the expressive power for Boolean connectivity graphs. Where binary decision diagrams have at most two outgoing edges, BCGs may have more than two. The recursion halts when the expression is empty. A detailed example of the algorithm is provided in Figure 7.

4.2.3 Undirected simple graph. Given the Boolean connectivity graph, we want to construct a new graph to satisfy Property I. The input is the Boolean connectivity graph, and the output is an undirected simple graph. The Boolean connectivity graph is a directed multigraph, which entails that there may be more than one edge between two pairs of nodes. To construct the simple graph, we will split nodes into new nodes, and connect these newly constructed nodes. More specifically, given two pairs of nodes u and v with k edges between them, we will construct $\delta(u)-1$ and $\delta(v)-1$ duplicate nodes respectively, such that $\delta(u) \times \delta(v) \geq k$. Then, the duplicate nodes and the original node are connected using an edge with Boolean truth value 1 (*true*).

Algorithm 1 Expression tree to Boolean connectivity graph

```
1:  procedure CONVERT(expression)
2:      if expression.operation == None then
3:          return Graph(expression.literal)
4:      end if
5:      left ← Convert(expression.left)
6:      right ← Convert(expression.right)
7:      if expression.operation == ∧ then
8:          Merge(left.output, right.input)
9:          return Graph(left, right)
10:     else if expression.operation == ∨ then
11:         Merge(left.input, right.input)
12:         Merge(left.output, right.output)
13:         return Graph(left, right)
14:     end if
15: end procedure
```

In Algorithm 2, we provide an algorithm to determine the number of duplicate nodes that must be constructed for each node to satisfy the condition. Initially, we set $\delta = 1$ for all nodes. Subsequently, we iterate over all edges between every pair of nodes. The algorithm tries to evenly distribute the duplicate nodes by incrementing δ for either u or v as long as $\delta(u) \times \delta(v) < k$. In Figure 8(a), we show the values of δ for three nodes, in (b) the duplication of nodes, and in (c) the resulting simple graph.

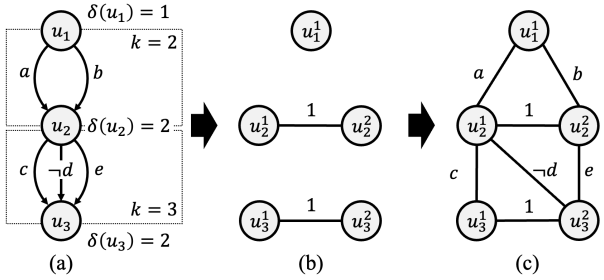


Figure 8: Construction of a simple undirected graph from a directed multigraph.

Algorithm 2 Computing the number of duplicate nodes

```
1:  for node in graph.nodes do
2:      δ(node) = 1
3:  end for
4:  for node1 in graph do
5:      for node2 in graph do
6:          k ← graph.CountEdges(node1, node2)
7:          while δ(node1) × δ(node2) < k do
8:              if δ(node1) < δ(node2) then
9:                  δ(node1) ← δ(node1) +1
10:             else
11:                 δ(node2) ← δ(node2) +1
12:             end if
13:         end while
14:     end for
15: end for
```

Table 1: Comparison of the graph properties (nodes and edges) and the hardware resources for the crossbar design (rows, columns, semiperimeter, area, and number of non-zero memristors) for both multiple single-output Boolean functions and a single multi-output Boolean function.

Benchmark	Multiple single-output Boolean functions								Single multi-output Boolean function							
	Graph		Crossbar design						Graph		Crossbar design					
	Nodes (num)	Edges (num)	Rows (num)	Cols (num)	Semi (num)	Area (num)	Mems (num)	Time (s)	Nodes (num)	Edges (num)	Rows (num)	Cols (num)	Semi (num)	Area (num)	Mems (num)	Time (s)
cm150a	32	47	22	27	49	594	47	0.34	33	48	12	21	33	252	48	0.019
t481	44	70	23	31	54	713	64	0.265	45	71	31	25	56	775	65	0.153
x2	52	66	24	36	60	864	60	0.893	36	62	19	27	46	513	58	0.016
cm163a	77	88	33	48	81	1584	73	0.841	44	73	24	33	57	792	68	0.017
misex1	76	96	37	47	84	1739	92	0.623	37	66	23	26	49	598	61	0.02
cordic	97	152	63	58	121	3654	145	0.343	97	154	40	70	110	2800	147	36.8
5xp1	124	175	61	84	145	5124	161	1.09	105	164	56	83	139	4648	160	0.024
clip	159	251	89	112	201	9968	231	0.531	165	250	93	116	209	10788	241	0.029
alu4	1189	1706	648	800	1448	518400	1649	15.336	1296	1834	697	863	1560	601511	1813	3.12
misex3	1245	1864	636	828	1464	526608	1812	8.637	752	1220	418	540	958	225720	1194	0.89
apex2	390	571	217	257	474	55769	546	39.471	415	599	227	280	507	63560	578	1.27
apex4	4206	5525	2241	2797	5038	6268077	5479	8.2	1402	3120	813	1030	1843	837390	2840	5.22
apex5	2117	3003	1136	1302	2438	1479072	2652	11.383	1851	2911	964	1198	2162	1154872	2740	0.117
seq	2733	3665	1408	1742	3150	2452736	3557	37.11	1459	2202	749	1016	1765	760984	2165	0.539
Geomean	271.0	380.9	143.9	180.9	325.2	26017.6	357.2	2.3	204.5	328.8	108.6	141.9	251.5	15402.9	313.8	0.2
Ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**0.8**	**0.9**	**0.8**	**0.8**	**0.8**	**0.6**	**0.9**	**0.1**

4.2.4 Node labeling. To satisfy Property II from Section 4, we must convert the undirected simple graph into an undirected bipartite graph. In previous work, a labeling algorithm was proposed where nodes are labeled with V, H, or VH, indicating whether they will be assigned to vertical, horizontal, or both vertical and horizontal nanowires in the crossbar design [16]. An integer linear programming solution (ILP) was proposed which is NP-hard and thus time-consuming. We employ a faster and more practical greedy algorithm to label the nodes with V, H, or VH based on the constraints in [16].

4.3 Crossbar construction

The final step is the construction of a crossbar design based on the labeled undirected simple graph. In [16], an analogy was defined between a the graph of a binary decision diagram and a memristor crossbar. In this analogy, the nodes are assigned to the wordlines and bitlines in a crossbar according to their label V, H, or VH. The edges with the literals as labels are assigned to memristors in a crossbar. For this last step, we use the same analogy to construct the crossbar design using the labeled undirected simple graph.

5 MULTI-OUTPUT BOOLEAN FUNCTIONS

Real-world combinational functions typically have multiple outputs. Sharing logic among multiple Boolean functions may result in smaller crossbar designs. Therefore, we propose a multi-step evaluation methodology based on multiplexing for multi-output Boolean functions. More specifically, for each output function f_i, we introduce a selector variable s_i. Then the overall Boolean function is constructed as a disjunction of the selector variable s_i conjoined with the function f_i, as shown below:

$$f_1 = (a \vee b) \wedge c \tag{1}$$

$$f_2 = a \vee b \tag{2}$$

$$f = (s_1 \wedge f_1) \vee (s_2 \wedge f_2) \tag{3}$$

The Boolean function f (line 3) then follows the same synthesis method as in Section 4. Evaluation of the multi-output Boolean function with N outputs is then accomplished using N steps. In each step $i, i \in [1, .., N]$, only the selector variables s_i are set to 1 while the other selector variables $s_j, j \neq i$, are set to 0. In Figure 9, we illustrate the evaluation of a multi-output Boolean function.

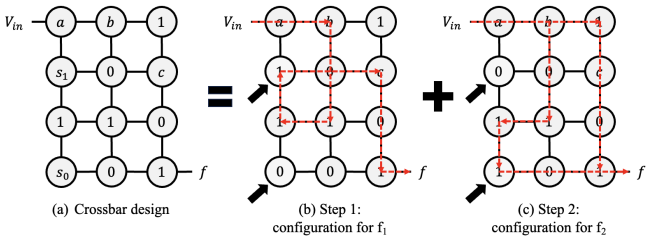


Figure 9: Multi-step evaluation for a multi-output Boolean function with outputs f_1 and f_2. For each step, we have highlighted the paths for the respectively Boolean functions using dashed lines in red. The paths for both outputs are different due to the different state of the memristors with variables s_1 and s_2. For f_1, the selector variables are set to $s_1 = 1$ and $s_2 = 0$, and for f_2 to $s_1 = 0$ and $s_2 = 1$.

6 EXPERIMENTAL RESULTS

In this section, we will evaluate our proposed FACTOR framework and compare the framework with other digital in-memory computing paradigms. The code for FACTOR is written in Python and the experiments are conducted on a machine with a 12th Gen Intel® Core™ i7-12700 × 20 processor. The code is on GitHub[1]. The experiments are conducted on 14 benchmarks from RevLib [18].

6.1 Evaluation of the FACTOR framework

First, we evaluate the proposed FACTOR framework in terms of graph properties and hardware resources. The graph properties are the number of nodes and edges for the Boolean connectivity graph. The hardware resources are the number of rows, columns, semiperimeter (rows+columns), area (rows×columns), the number of non-zero memristors, and the synthesis time. In Table 1, we compare the graph properties and hardware resources for the crossbar design for two approaches. In the first approach, the benchmarks are synthesized as multiple single-output Boolean functions and in the second approach as a single multi-output Boolean function. We observe that the number of nodes and the number of edges is reduced by 20% and 10% when the logic is shared. Consequently,

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

Table 2: Comparison of the hardware resources in terms of rows, columns, semiperimeter, area, non-zero memristors, and synthesis time for different synthesis methods for flow-based computing.

Benchmark	Chakraborty et al. [4]						COMPACT [16]						FACTOR					
	Rows (num)	Cols (num)	Semi (num)	Area (num)	Mems (num)	Time (min)	Rows (num)	Cols (num)	Semi (num)	Area (num)	Mems (num)	Time (min)	Rows (num)	Cols (num)	Semi (num)	Area (num)	Mems (num)	Time (min)
cm150a	32	48	80	1536	48	0.0	12	22	34	264	48	0.0	22	27	49	594	47	0.0
t481	32	37	69	1184	58	0.0	17	23	40	391	58	0.0	23	31	54	713	64	0.0
x2	52	74	126	3848	89	0.0	33	35	68	1155	89	0.1	19	27	46	513	58	0.0
cm163a	66	90	156	5940	92	0.1	25	31	56	775	78	0.0	24	33	57	792	68	0.0
misex1	72	92	164	6624	101	0.0	21	29	50	609	72	0.0	23	26	49	598	61	0.0
cordic	99	131	230	12969	170	0.0	42	44	86	1848	142	0.1	63	58	121	3654	145	0.0
5xp1	117	155	272	18135	185	0.1	52	53	105	2756	162	0.1	56	83	139	4648	160	0.0
clip	148	186	334	27528	253	0.0	84	84	168	7056	253	0.0	89	112	201	9968	231	0.0
alu4	1281	1461	2742	1871541	2299	0.4	683	686	1369	468538	2299	1.5	648	800	1448	518400	1649	0.3
misex3	1552	1670	3222	2591840	2486	0.4	674	676	1350	455624	2292	2.2	418	540	958	225720	1194	0.0
apex2	1644	1645	3289	2704380	2759	1.4	909	936	1845	850824	2759	1.1	217	257	474	55769	546	0.7
apex4	1644	1789	3433	2941116	2912	0.4	508	528	1036	268224	1910	2.3	813	1030	1843	837390	2840	0.1
apex5	2674	3591	6265	9602334	4352	1.0	1409	1497	2906	2109273	4352	0.9	1136	1302	2438	1479072	2652	0.2
seq	3231	3690	6921	11922390	4982	1.5	1743	1778	3521	3099054	4982	1.1	749	1016	1765	760984	2165	0.0
Geomean	282.0	348.6	631.6	98307.5	458.7	0.0	128.6	145.5	275.1	18709.5	417.4	0.0	114.6	143.6	258.6	16459.9	307.6	0.0
Ratio	1.0	1.0	1.0	1.0	1.0	**1.0**	0.5	0.4	0.4	0.2	0.9	2.6	**0.4**	**0.4**	**0.4**	**0.2**	**0.7**	1.8

the required hardware resources are also lower on average with an average reduction of 20% for the semiperimeter, 40% for the area, and 10% for the number of non-zero memristors.

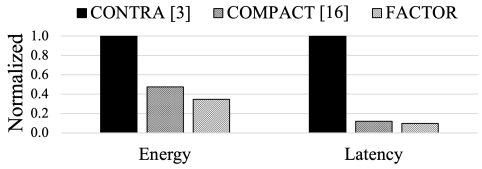


Figure 10: Comparison of average energy consumption and latency over fourteen Revlib benchmarks for CONTRA [3], COMPACT [16], and our proposed framework FACTOR.

6.2 Comparison for digital in-memory computing

In this section, we compare our proposed FACTOR framework with other frameworks for digital in-memory computing. First, we compare with previous work for flow-based computing, and then we compare with other digital in-memory computing paradigms.

In Table 2, we compare our proposed synthesis method with the previous state-of-the-art synthesis methods for flow-based computing [4, 16]. For FACTOR, we use the results with the least number of non-zero memristors from Table 1. We observe that our proposed method uses the least amount of resources with a reduction of 60% for the semiperimeter, 80% for the area, and 30% for the number of non-zero memristors. This is due to that previous work relies on BDDs as underlying data structure whereas FACTOR relies on Boolean connectivity graphs, which are more expressive.

Next, we evaluate the energy and latency for our proposed FACTOR framework for flow-based computing with CONTRA [3] for the MAGIC computing paradigm, and with COMPACT for flow-based computing. The READ/WRITE energy to program a ReRAM device is $1.08pJ$ and $3910pJ$, and the READ/WRITE latency is $29.31ns$ and $50.88ns$ [19]. In Figure 10, we show the normalized energy and latency over the 14 benchmarks for all three frameworks. We observe that our proposed FACTOR framework outperforms CONTRA and COMPACT in terms of energy and latency with an energy and latency reduction of 65% and 90% compared with CONTRA, and 12% and 2% with COMPACT. This is due to the low number of WRITE operations.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented FACTOR, a framework for synthesizing crossbar designs for flow-based computing. Our innovation lies in the exploration of data structures, called Boolean connectivity graphs, which are more succinct than binary decision diagrams. In contrasts with BDDs, BCGs are not limited to two outgoing edges for each internal node in the graph. This has a first-order impact on the overall energy efficiency and latency for the flow-based computing system. Using FACTOR, the semiperimeter (rows+columns) reduces by 60% and the number of non-zero memristors by 30% compared with the state-of-the-art. This results in an area, energy and latency reduction of 80% and 12%, and 2% compared with the state-of-the-art. In our future work, we will explore other computational structures that can supports flow-based computing systems.

REFERENCES

[1] Zahiruddin Alamgir et al. 2016. Flow-based computing on nanoscale crossbars: Design and implementation of full adders. In *ISCAS*. IEEE, 1870–1873.
[2] Frank Arute et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
[3] Debjyoti Bhattacharjee et al. 2020. Contra: area-constrained technology mapping framework for memristive memory processing unit. In *ICCAD*. 1–9.
[4] Dwaipayan Chakraborty and Sumit Jha. 2017. Automated synthesis of compact crossbars for sneak-path based in-memory computing. In *DATE*. IEEE, 770–775.
[5] Amad Ul Hassen. 2017. Automated synthesis of compact multiplier circuits for in-memory computing using ROBDDs. In *NANOARCH*. IEEE, 141–146.
[6] Amad Ul Hassen et al. 2018. Free binary decision diagram-based synthesis of compact crossbars for in-memory computing. *TCAS-II* 65, 5 (2018), 622–626.
[7] Shahar Kvatinsky et al. 2012. MRL—Memristor ratioed logic. In *International Workshop on Cellular Nanoscale Networks and their Applications*. IEEE, 1–6.
[8] Shahar Kvatinsky et al. 2013. Memristor-based material implication (IMPLY) logic: Design principles and methodologies. *IEEE VLSI* 22, 10 (2013), 2054–2066.
[9] Shahar Kvatinsky et al. 2014. MAGIC—Memristor-aided logic. *IEEE TCAS-II* 61, 11 (2014), 895–899.
[10] Gordon E. Moore. 1964. Cramming More Components onto Integrated Circuits. *Electronics* (1964).
[11] Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, Fan Yao, and Rickard Ewetz. 2022. Hybrid digital-digital in-memory computing. In *DATE*. IEEE, 1177–1180.
[12] Abu Sebastian et al. 2020. Memory devices and applications for in-memory computing. *Nature nanotechnology* 15, 7 (2020), 529–544.
[13] E.M. Sentovich et al. 1992. *SIS: A System for Sequential Circuit Synthesis*. Technical Report UCB/ERL M92/41. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html
[14] Bhavin J Shastri et al. 2021. Photonics for artificial intelligence and neuromorphic computing. *Nature Photonics* 15, 2 (2021), 102–114.
[15] Thomas N Theis and H-S Philip Wong. 2017. The end of moore's law: A new beginning for information technology. *Computing in science & engineering* 19, 2 (2017), 41–50.
[16] Sven Thijssen, Sumit Jha, and Rickard Ewetz. 2021. Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter. In *DATE*. IEEE, 232–237.
[17] Alvaro Velasquez and Sumit Kumar Jha. 2014. Parallel computing using memristive crossbar networks: Nullifying the processor-memory bottleneck. In *IDT*. IEEE, 147–152.
[18] Robert Wille et al. 2008. RevLib: An online resource for reversible functions and reversible circuits. In *ISMVL*. IEEE, 220–225.
[19] Tao Yang et al. 2021. PIMGCN: a reram-based PIM design for graph convolutional network acceleration. In *DAC*. IEEE, 583–588.