Technology-Aware Logic Synthesis for ReRAM based In-Memory Computing

Debjyoti Bhattacharjee*‡, Luca Amarú† and Anupam Chattopadhyay*
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†Synopsys, CA, USA. Corresponding author‡ : {*debjyoti001*}*@ntu.edu.sg*

Abstract— **Resistive RAMs (ReRAMs) have gained prominence for design of logic-in-memory circuits and architectures due to fast read/write speeds, high endurance, density and logic operation capabilities. ReRAM crossbar arrays allow constrained bit-level parallel operations. In this paper, for the first time, we propose optimization techniques during logic synthesis, which are specifically targeted for leveraging the parallelism offered by ReRAM crossbar arrays. Our method uses Majority-Inverter Graph (MIG) for the internal representation of the Boolean functions. The novel optimization techniques, when applied to the MIG, exposes the bit-level parallelism, and is further coupled with an efficient technology mapping flow. The entire synthesis process is benchmarked exhaustively over large arithmetic functions using a representative ReRAM crossbar architecture, while varying the crossbar dimensions. For the hard benchmarks, we obtained 10% reduction in the number of nodes with 16% reduction in delay on average.**

I. Introduction

Resistive RAM (ReRAM) has emerged as a promising technology for logic-in-memory computations, due to simulataneous storage and logic capabilities. ReRAMs offer high endurance and density, high energy efficiency and fast operating speeds. ReRAMs have been used for realization of application specific circuits such Boolean arithmetic circuits [1]–[3], ternary arithmetic [4] as well as neuromorphic computing [5], [6]. ReRAMs can realize a universal set of Boolean functions, which has been leveraged for developing general-purpose programmable architecture [7]–[9].

Multiple approaches for computation using ReRAM devices have been proposed in the recent years. A preliminary methodology for computing Boolean functions using memristive devices was presented by Lehtonen et al. [10]. Chattopadhyay et al. [11] proposed logic synthesis solution for memristors that realize material implication. For ReRAM devices realizing Boolean majority, delay optimal technology mapping method was proposed by Bhattacharjee et al. [12], followed by heuristics for area constrained technology mapping [13]. Logic synthesis techniques for minimizing the number of ReRAM devices and delay was proposed by Shirinzadeh et al. [14]. However, none of these these works addressed the tremendous possibility of parallel operations on ReRAM crossbar arrays. This deficit in the logic synthesis process naturally leads to a sub-optimal computing solution, as far as the ReRAM crossbar arrays is concerned. While technology-aware logic synthesis has been done for Quantum-dot Cellular Automata (QCA) in [15], [16], and for ReRAM device-level endurance management [17], to the best of our knowledge, none of these works have considered the opportunity to exploit bit-level parallelism available in crossbar arrays.

In accordance with ReRAM device operations and state-of-the-art circuit design studies [18], it is well known that ReRAM crossbar operations inherently allows parallel computations on all the devices that share a wordline. In a single cycle, all devices with a common wordline can be read out simultaneously and the device states' do not change during a read operation. ReRAM devices, which do not share wordlines cannot be read simultaneously. A read-out value can be applied to any wordline and zero or more bitlines. Considering these crossbar constraints, the ReVAMP architecture was proposed [9], along with a method for generating instructions for the same. In this work, we extend in this direction by proposing a novel logic synthesis technique that optimizes the logic network to harness the inherent parallelism of the ReRAM crossbar arrays. Our contributions in this work are the following.

- Novel logic synthesis techniques considering crossbar constraints, which can be applied, in principle, to any in-memory computing technology.
- Combined synthesis and technology mapping flow for harnessing the bit-level parallelism.
- Experimental study using representative ReRAM crossbar architecture and complex benchmark functions.

The rest of the paper is organized as follows. Section II presents the ReVAMP architecture along with its instruction set and the basics of logic representation using Majority-Inverter Graphs. In section III, we describe the novel logic synthesis techniques considering crossbar constraints. Section IV presents the benchmarking results of the proposed synthesis method against baseline synthesis method. Finally, section V presents conclusion to the paper and presents directions of future research.

II. Preliminaries

A. ReVAMP Architecture

ReRAM crossbar memory consists of multiple 1S1R ReRAM devices [19], arranged in the form of a crossbar array [18]. Each ReRAM device has two input terminals — the wordline wl and the bitline bl. The internal resistive state Z of the ReRAM acts as a third input and the stored bit. The next

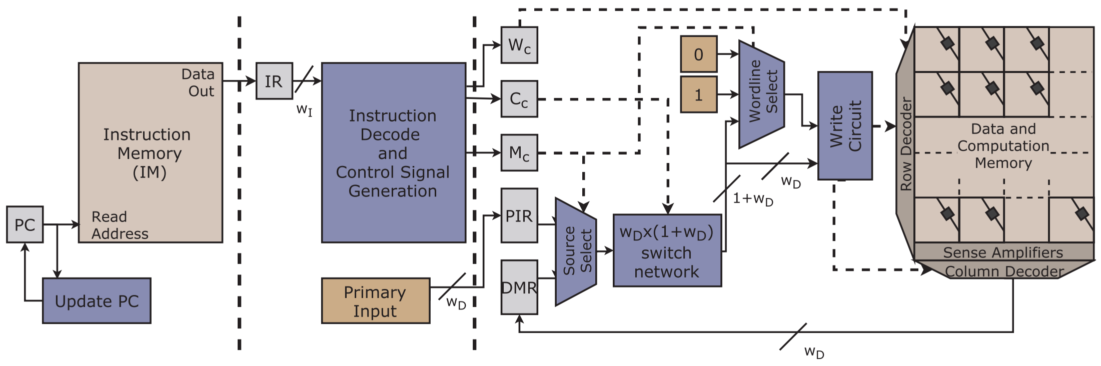

Fig. 1: *ReVAMP architecture*

state of the device Z_n can be expressed as a 3-input Boolean majority function, with the bitline input inverted.

$$Z_n = M_3(Z, wl, \overline{bl}) \tag{1}$$

This forms the fundamental logic operation that can be realized using ReRAM devices. The inversion operation can be realized using the intrinsic function Z_n. Since majority and inversion operations form a *functionally complete* set, any Boolean function can be realized using only Z_n operations. Like conventional RAM arrays, ReRAM memories are accessed as words.

In [9], authors proposed the ReRAM based ReVAMP architecture that utilizes two ReRAM crossbar memories — the Instruction Memory (IM) and the Data Storage and Computation Memory (DCM). The IM is a regular instruction memory accessed using the program counter (PC). The DCM is used for data storage as well as for in-memory computation. A three-stage pipeline processes the instructions.In the Instruction Fetch (IF) stage, the instruction at the address held by the Program Counter (PC) is fetched from the IM and loaded into the instruction register (IR), before the PC is updated. In the decode stage, the instruction available in IR is decoded to determine the control inputs for the wordline source select multiplexer, crossbar interconnect network and the write circuit.

The primary input register (PIR) buffers the primary input data while the data memory register (DMR) stores the word read out from the DCM. Both PIR and DMR are w_D bits wide, i.e. same as the number of bitlines in the DCM. Depending on the control input M_c, the source select multiplexer selects either PIR or the DMR as the data source. Thereafter, the crossbar-interconnect network generates the wordline and w_D number of bitline inputs by appropriate permutation of the selected data source, as per the control signals stored in C_c.

The write circuits reads the value of the target wordline from the W_c register and the output of the crossbar-interconnect to determine and apply the inputs to the row and column decoder of the DCM.

ReVAMP Instruction Set: The ReVAMP architecture supports two instructions—*Read* and *Apply*, in the formats shown in Fig. 2. The *Read* instruction reads the word at the address **wl** from the DCM and stores it in the DMR. Now available in the DMR, this word can be used as input by the next instruction.

Read	w

Apply	w	s	ws	wb	(v val$_1$)	(v val$_2$)	…	(v val$_{w_D}$)

Fig. 2: ReVAMP instruction format

The *Apply* instruction uses the DCM for computation.The word address **w** selects the word in the DCM that will be computed upon. A source select flag **s** chooses whether the inputs will be be primary input (PIR) or read out word (DMR). A two bit flag **ws** specifies the worline input — 00 selects Boolean 0, 01 selects Boolean 1 and 11 selects input specified by the **wb** flag. The **wb** bits are used to specify the bit within the chosen data source for use as wordline input. Pairs **(v,val)** are used to specify bitline inputs. Valid bit **v** indicates if the input is NOP or a valid input. Similar to **wb**, bits **val** specifies the bit within the chosen data source to be used as bitline input.

B. Brief Background on MIG

A Majority-Inverter-Graph (MIG) is a data structure for Boolean function representation and optimization that consists of 3-input Boolean majority nodes and regular/complemented edges. The expressive power of the majority operator allows efficient representation of Boolean functions using MIGs. Traditional AND/OR/INV graphs (AOIGs) are a special case of MIGs. This is because in the case of 3-input majority operator, fixing one input to 0 realizes an AND while fixing one input to 1 realizes an OR. It is feasible to derive MIGs from AOIGs but fully exploiting the majority functionality, i.e., with non-constant inputs leads to smaller MIG representations. Due to the native realization of Majority operation in ReRAM devices, MIG is used heavily in synthesis [14] and technology mapping [12] flows for ReRAM crossbar array.

III. Crossbar-aware MIG Optimization Methodology

In this section, we present the optimization techniques on MIGs that are tuned to crossbar constraints. In addition, we also review the crossbar-aware compilation technique used to generate ReVAMP instructions from the MIG representation of the Boolean network.

A. MIG Algebraic Optimization

We are interested in compact MIG representations because they translate in smaller and faster physical implementations, especially in the ReRAM nanotechnology. In order to manipulate MIGs and reach advantageous MIG representations, a dedicated Boolean algebra was introduced in the original paper [20]. The axiomatic system for the MIG Boolean algebra, referred to as Ω, is defined by the five following primitive transformation rules.

Commutativity — $\Omega.C$

$$M(x,y,z) = M(y,x,z) = M(z,y,x)$$

Majority — $\Omega.M$

$$if(x = y) : M(x,y,z) = x = y$$
$$if(x = \overline{y}) : M(x,y,z) = z$$

Associativity — $\Omega.A$

$$M(x,u,M(y,u,z)) = M(z,u,M(y,u,x))$$

Distributivity — $\Omega.D$

$$M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z)$$

Inverter Propagation — $\Omega.I$

$$\overline{M}(x,y,z) = M(\overline{x},\overline{y},\overline{z})$$

Some of these axioms are inspired from median algebra [21], [22] and others from the properties of the median operator in a distributive lattice [23]. A strong property of MIGs and their algebraic framework is about reachability. It has been proved that, by using a sequence of transformations drawn from Ω, it is possible to traverse the entire MIG representation space. Unfortunately, deriving a sequence of Ω transformations is an intractable problem. As for traditional logic optimization, heuristic techniques provide here fast solutions with reasonable quality. Details of the optimization algorithms targeting depth, size and power reductions in an MIG can be found in [20] .

B. MIG Boolean Optimization

MIG optimization techniques based on Ω rules are called *algebraic*. More powerful transformations are possible by using Boolean methods. For the purposes of this work, we focus on MIG Boolean optimization based on Input Partitioning Methods (IPM) and safe errors.

Safe Errors: MIGs are hierarchical majority voting systems. One notable property of majority voting is the ability to correct different types of bit-errors. This feature is inherited by MIGs, where the error masking property can be exploited for logic optimization. The idea is to purposely introduce logic errors that are succesively masked by the voting resilience in MIG nodes. For example, rewrite an MIG node w as $w = M(wA, wB, wC)$, where A, B, C are errors on node w. If such errors are advantageous, in terms of logic simplifications, better MIG representations can be generated. To be considered safe in an MIG, logic errors must be pairwise orthogonal.

Input Partitioning Methods: The rationale behind IPM for MIG is to select inputs leading to advantageous simplifications when erroneous. For this purpose, we use a decision metric, called dictatorship, to select the most profitable inputs for logic error insertion. The dictatorship is the ratio of input patterns over the total (2^n) for which the output assumes the same value than the selected input. For example, in the function $f = (a + b)c$, the inputs a and b have equal dictatorship of $5/8$ while input c has an higher dictatorship of $7/8$. The inputs with the highest dictatorship are the ones where we want to insert logic errors. Indeed, they have the largest influence on the circuit functionality and its structure. Exact computation of the dictatorship requires exhaustive simulation of an MIG structure, which is not feasible for practical reasons. Heuristic approaches to estimate dictatorship involve partial random simulation and graph techniques.

After exact or heuristic computation of the dictatorship, we select a subset of the primary inputs with highest dictatorship. Next, for each selected input, we determine a condition that causes an error. We require these errors to be orthogonal. Since we operate directly on the primary inputs, we already divide the Boolean space into disjoint subsets that are orthogonal. Because we need at least three errors, we need to consider at least three inputs to be made erroneous, say x, y and z. A possible partition is the following: $\{x! = y, x = y = z, x = y = z'\}$. The corresponding errors are $A : x = y$ for $\{x! = y\}$, $B : z = y'$ when $x = y$ for $\{x = y = z\}$ and $C : z = y$ when $x = y$ for $\{x = y = z'\}$. We refer the reader to [20] for a formal proof on A, B and C orthogonality.

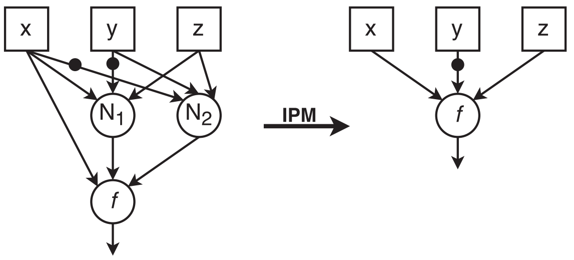

Fig. 3: IPM for simplification of MIG

We illustrate input partitioning method on a function $f = M(x, M(x, y', z), M(x', y, z))$. Corresponding to f, the input partition is $\{x! = y, x = y = z, x = y = z'\}$ which is affected by errors A, B and C, respectively. The first error A imposes $x = y$ which leads to $fA = M(x, M(y, y', z), M(x', x, z))$. This can be simplied by using $\Omega.M$ to $fA = M(x, z, z) = z$. The second error B imposes $z = y'$ when $x = y$. This is the case for the bottom level majority operators $M(x, y', z)$ and $M(x', y, z)$ which are transparent when $x = y$. Therefore, error B leads to $fB = M(x, M(x, y, y'), M(x', y, y'))$ which can be further simplified to $fB = M(x, y', x') = y'$ by applying $\Omega.M$. The third error C imposes $z = y$ when

$x = y$ holds. Analogously to error B, error C leads to $fC = M(x, M(x, y, y), M(x, y, y))$ which can be further simplified into $fC = M(x, x, y) = x$ by $\Omega.M$. A top majority node finally merges the three functions into $f = M(fA, fB, fC) = M(z, y', x)$ which correctly represents the objective function but has 2 fewer nodes and 1 level less than the original representation.

Safe error insertion in MIGs can be used for size reduction. In this case, the branch triplication overhead in $w = M(wA, wB, wC)$ imposes tight simplification requirements. We use the input partitioning method to focus on the most influent inputs of a MIG, and introduce selective simplification on them.

Majority Refactoring: An effective Boolean technique for MIG minimization is majority refactoring. The idea is to first compute a large cone of logic rooted at a MIG node. This cone of logic is transformed into a canonical logic representation, e.g., a BDD or a truth table. Starting from the canonical logic representation, a new local MIG is derived by means of decomposition. Novel techniques for majority decomposition have been recently proposed [24], [25], which can be used in conjunction with state-of-the-art solutions [26]. Once the new local MIG is formed, its cost is calculated and compared to the original MIG, where the cone of logic was extracted. If an advantage in the cost metric used is seen, the refactored MIG is imported back to the global network. This procedure can be iterated for every node of the MIG in topological order, and then until an improvement is seen (or a maximum runtime limit is hit).

Now, we discuss the cost metrics used for MIG optimization. The number of nodes in an MIG is a cost metric of the number of computations needed. So, reduction of number of nodes in MIG is one of our synthesis objectives. In synthesis of traditional CMOS circuits, depth of the logic representation structure is an indication of the delay of the synthesized circuit. However, in case of mapping to ReRAMs, this might not hold true, since all the operations in a level of the logic networks might not be computed in parallel under the crossbar constraints. In Figure 5, we show an MIG with w as a common input for nodes R_1, R_2, R_3 and R_4. There is a predecessor b_i for node R_i which is connected with an inverted edge. Such an MIG can be computed in a single step if all the nodes S_i are stored in a word and w is applied as wordline input and corresponding b_i as bitline inputs. Based on this observation, the rationale for crossbar-aware optimization is to: *first*, enforce logic sharing in the MIG already starting from the lower levels of the network, and *second*, share non-inverted edges, if multiple inverted-edges are shared, propagate the inverts above. With this background, we have tailored the previous MIG techniques for crossbar-aware logic optimization.

We created a new MIG optimization flow, targeting crossbar-aware logic realizations, comprising of MIG algebraic rewriting, MIG Boolean IPM and MIG majority refactoring. All three techniques have been made aware of the optimization

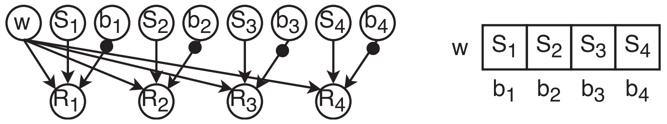

Fig. 5: Shared input w can be used as wordline input.

goals convenient to crossbar ReRAMs.

A high level execution flow is shown in Figure 4. The process is typically iterated less than 10 times ($n < 10$). In practice, we found that good size reductions, with controlled net count increase and no depth overhead, produce the most advantageous MIG representation for later mapping into the ReRAM array.

C. Compilation Method

Once the cross-aware logic optimizations have been performed on the MIG, the MIG has to be realized using the ReVAMP instructions. To do so, we use the four stage compilation strategy proposed by Bhattacharjee et al. [9] and shown in Figure 4. We briefly present the four step compilation method.

1) *Assign host and inputs to nodes:* The computation of an MIG node n has to be hosted by a device, that stores one of the predecessor p of node n. The other two predecessor of the node can act as either the bitline or wordline input. The assignment of hosts and inputs is done such that computations of multiple nodes can be done in parallel, if they share common predecessors. This assignment of host and input is marked on the edges of the MIG.
2) *Group nodes into blocks:* Due to the crossbar constraints (only a single wordline can be read out in a cycle), the wordline and bitline inputs of a node must reside on the same wordline. This group of input nodes is termed as a *block*. Since a block has to fit within a single wordline, the number of nodes grouped in a block must not be greater than the word length w_D of the DCM. Blocks can be merged if they have common inputlines with the merged block having only a single copy of common inputlines. Further, blocks in the same level that have hosts which share a wordline input are merged to allow the possibility of parallel computation.
3) *Pack blocks into words:* The blocks enforce which nodes have to be placed in the same wordline. It is possible that the total number of elements in multiple blocks is less than the word length of the DCM. In that case, all these blocks can be packed together in a single wordline. Based on this intuition, the goal is to pack the blocks into minimum number of words.
4) *Generate instructions:* In the final step of the compilation method, we determine the ReVAMP instructions. Each computation is expressed as an *Apply* instruction and read operations are expressed as *Read* instruction. The nodes in level i are scheduled for computation before any node at level $i + 1$ is scheduled to respect data dependencies between nodes. The hosts grouped in the same block as scheduled for computation together.

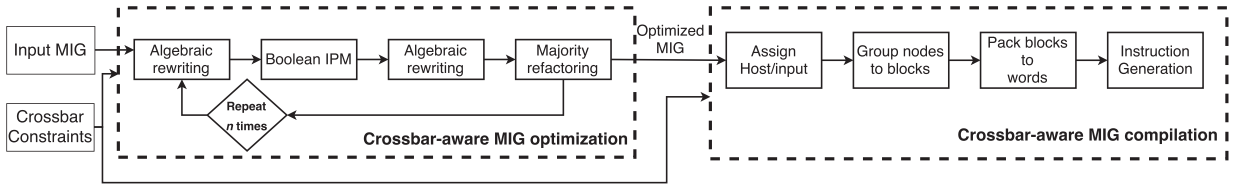

Fig. 4: MIG optimization flow for crossbar-aware logic synthesis.

Once all the nodes in a level have been computed, we determine whether any inverted copies of the nodes are required for computation of nodes at a higher level. If inverted copies are needed, the node stored in inverted form in the required block by reading it out, followed by writing it through the bitline.

In the next section, we present the results of benchmarking the proposed crossbar aware end-to-end flow from MIG to generation of instructions.

IV. Experimental Results

The proposed end-to-end flow is implemented for the ReVAMP architecture. To evaluate our optimization techniques, we choose the depth-optimized hard EPFL benchmarks, available as MIGs [1]. The results are shown in Table I for DCM word length $w_D = 16$. We explain the results by means of a benchmark, say *adder*. The initial MIG has 3369 nodes along with 9333 edges while the optimized MIG has 2811 nodes along with 7662 edges. The reduction in number of nodes might not directly lead to reduction in delay of mapping. The quality of the MIG for mapping can be determined by the number of blocks required to group the nodes, and the number of words which are required to pack the blocks into words. For the *adder* benchmark, the optimized MIG requires 27.4% less blocks with 31% reduction in number of words. Finally, we can see that these optimizations lead to 39.6% reduction in overall delay of mapping.

In general, we have managed to reduce the number of nodes as well as edges for all the benchmarks. This in turn has lead to reduction in the number of blocks, thereby leading to the reduction of the number of words required for the mapping. On average, we have been able to reduce the number of nodes by 10.8% (maximum reduction of 16.56% for adder) along with the average overall delay reduced by 16.67% (maximum reduction of 39.64% for adder). In addition, our experiments show that the packing acheives more than 99% device utlization. This clearly demonstrates the overall efficiency of our crossbar aware MIG optimization in reducing both the number of devices required as well as mapping delay.

The word length of the DCM determines the number of parallel operations that can be performed. Therefore, we analyse the impact of word length on our flow using another set of EPFL benchmarks. The results of mapping on varying word length w_D from 32 to 256 is shown in Table II. With increase in word length, the overall mapping delay reduces for all the benchmarks, showing the effectiveness of the compilation flow to harness the parallelism offered by the DCM. Also for these benchmarks, our MIG optimization method reduces the number of nodes in the MIG. Figure 6 shows the percentage reduction in delay for the optimized MIGs compared to the base MIGs. With increase in word length, our optimized MIGs are better suited for mapping to the crossbar compared to the base MIGs. Figure 7 shows the percentage reduction in number of words compared to base MIGs. For all the word lengths, our optimized MIGs always require fewer number of words compared to the base MIGs.

[1]http://lsi.epfl.ch/benchmark

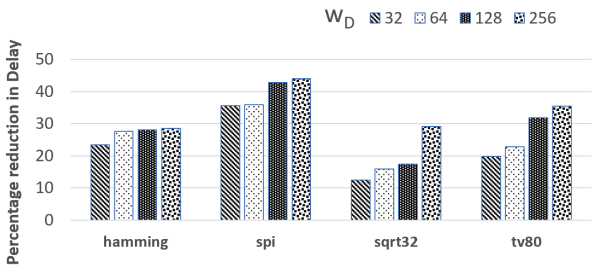

Fig. 6: Impact of crossbar size on delay

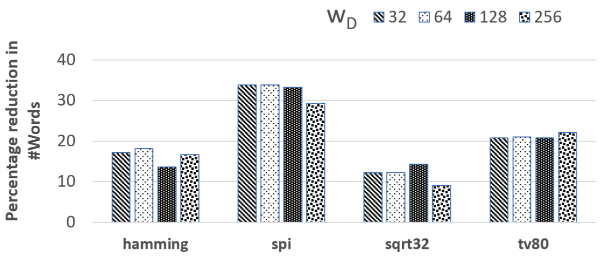

Fig. 7: Impact of crossbar size on #Words

V. Conclusion and Future Work

In-memory computing fabrics are getting major research attention nowadays, thanks to inherent advantage of tremendous boost in data locality. Traditional design automation flows are not designed for such computing architectures and therefore, poses a challenge towards adoption. Recent works in this direction have focused on logic synthesis and technology mapping with the capabilities of individual devices in mind. In this, we propose novel logic synthesis techniques, which, for the first time, enables one to exploit the bit-level parallelism available in ReRAM crossbar arrays. Our results show that it improves the area and delay metrics compared to a crossbar-agnostic design automation flow.

Looking forward, we intend to integrate multiple performance objectives, notably area, delay and device-level en-

TABLE I: Synthesis Results for hard EPFL benchmarks ($w_D = 16$), δ is % reduction computed as $\frac{unoptimized-optimized}{unoptimized} \times 100$

Benchmark	#Nodes		#Edges		#Blocks		#Words		Delay	
	Unopt.	Opt. (δ)	Unopt.	Opt. (δ)	Unopt.	Opt. (δ)	Unopt.	Opt. (δ)	Unopt.	Opt.(δ)
adder	3369	2811(16.6)	9333	7662(17.9)	669	486(27.4)	255	176(31.0)	12597	7603(39.6)
div	76005	69478(8.6)	227625	207834(8.7)	12935	12286(5.0)	5475	4945(9.7)	303585	265795(12.4)
log2	37650	36359(3.4)	112848	108963(3.4)	7546	7330(2.9)	2714	2617(3.6)	129851	124916(3.8)
max	7846	6108(22.2)	21996	16500(25.0)	1153	946(18.0)	428	323(24.5)	16896	11575(31.5)
mult	42194	34615(18.0)	126192	103458(18.0)	9262	8824(4.7)	3418	2667(22.0)	139236	107886(22.5)
sin	7946	7218(9.2)	23760	21534(9.4)	1509	1419(6.0)	629	566(10.0)	31075	27147(12.6)
sqrt	52538	49553(5.7)	157224	148182(5.8)	9525	9211(3.3)	3744	3525(5.8)	208948	192524(7.9)
square	19395	18838(2.9)	57987	56178(3.1)	5480	5560(-1.5)	1593	1537(3.5)	70678	68583(3.0)

TABLE II: Synthesis Results for small EPFL benchmarks across multiple w_D

Benchmark	Nodes		w_D=32				w_D=64				w_D=128				w_D=256			
			Delay		Words		Delay		Words		Delay		Words		Delay		Words	
	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.
div16	4440	3875	14088	10764	164	126	11186	7815	81	62	8193	5196	40	31	5941	3811	20	16
hamming	2280	2080	6007	4598	87	72	4824	3495	44	36	3767	2708	22	19	3045	2175	12	10
spi	3889	3865	7696	4953	130	86	6501	4167	65	43	5729	3281	33	22	5004	2803	17	12
sqrt32	2206	2082	7182	6293	82	72	5908	4971	41	36	4458	3685	21	18	3810	2699	11	10
tv80	8176	8025	21305	17089	283	224	18249	14087	143	113	16007	10916	72	57	13237	8553	36	28

durance of ReRAM crossbar array in a single design automation flow. An exact synthesis flow for the same is also an interesting open research problem.

REFERENCES

[1] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 5, no. 1, pp. 64–74, 2015.

[2] D. Bhattacharjee and A. Chattopadhyay, "Efficient binary basic linear algebra operations on reram crossbar arrays," in *VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID), 2017 30th International Conference on*, pp. 277–282, IEEE, 2017.

[3] D. Bhattacharjee, A. Siemon, E. Linn, and A. Chattopadhyay, "Efficient complementary resistive switch-based crossbar array booth multiplier," *Microelectronics Journal*, vol. 64, pp. 78–85, 2017.

[4] W. Kim, A. Chattopadhyay, A. Siemon, E. Linn, R. Waser, and V. Rana, "Multistate memristive tantalum oxide devices for ternary arithmetic," *Scientific reports*, vol. 6, 2016.

[5] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa and W. Lu, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano Letters*, vol. 12, no. 1, pp. 389–395, 2011.

[6] D. B. Strukov, D.R. Stewart, J. Borghetti, X. Li, M. Pickett, G. M. Ribeiro, W. Robinett, G. Snider, J. P. Strachan, W. Wu, Q. Xia, J. J. Yang and R. S. Williams, "Hybrid cmos/memristor circuits," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1967–1970, 2010.

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

[8] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1718–1725, EDA Consortium, 2015.

[9] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "Revamp: Reram based vliw architecture for in-memory computing," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 782–787, IEEE, 2017.

[10] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *NanoArch*, pp. 33–36, IEEE Computer Society, 2009.

[11] A. Chattopadhyay and Z. E. Rakosi, "Combinational logic synthesis for material implication," in *IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, VLSI-SoC 2011, Kowloon, Hong Kong, China*, pp. 200–203, 2011.

[12] D. Bhattacharjee and A. Chattopadhyay, "Delay-optimal technology mapping for in-memory computing using reram devices," in *Proceedings of the 35th International Conference on Computer-Aided Design*, p. 119, ACM, 2016.

[13] D. Bhattacharjee, A. Easwaran, and A. Chattopadhyay, "Area-constrained technology mapping for in-memory computing using reram devices," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC*, 2017.

[14] S. Shirinzadeh, M. Soeken, P. E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *Proceedings of DATE*, 2016.

[15] M. G. A. Martins, V. Callegaro, F. S. Marranghello, R. P. Ribas, and A. I. Reis, "Majority-based logic synthesis for nanometric technologies," in *14th IEEE International Conference on Nanotechnology*, pp. 256–261, Aug 2014.

[16] K. Kong, Y. Shang, and R. Lu, "An optimized majority logic synthesis methodology for quantum-dot cellular automata," *IEEE Transactions on Nanotechnology*, vol. 9, pp. 170–183, March 2010.

[17] S. Shirinzadeh, M. Soeken, P. E. Gaillardon, G. D. Micheli, and R. Drechsler, "Endurance management for resistive logic-in-memory computing architectures," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1092–1097, March 2017.

[18] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger and R. Waser, "Beyond von neumann-logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, 2012.

[19] A. Siemon, S. Menzel, A. Marchewka, Y. Nishi, R. Waser, and E. Linn, "Simulation of TaO$_x$-based complementary resistive switches by a physics-based memristive model," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pp. 1420–1423, IEEE, 2014.

[20] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimi tion," in *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6, ACM, 2014.

[21] H.-J. Bandelt and J. Hedlíková, "Median algebras," *Discrete mathematics*, vol. 45, no. 1, pp. 1–30, 1983.

[22] D. E. Knuth, *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Pearson Education India, 2011.

[23] S. Kiss and B. A. Math, "A ternary operation in distributive lattices," *Selected Papers on Algebra and Topology by Garrett Birkhoff*, p. 107, 1987.

[24] R. Devadoss, K. Paul, and M. Balakrishnan, "Majsynth: An n-input majority algebra based logic synthesis tool for quantum-dot cellular automata," in *Proceedings of IWLS*, 2015.

[25] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Bds-maj: A bdd-based logic synthesis tool exploiting majority logic decomposition," in *Proceedings of the 50th Annual Design Automation Conference*, p. 47, ACM, 2013.

[26] V. Varshavskii, "Compatible majority decomposition," *Cybernetics and Systems Analysis*, vol. 4, no. 3, pp. 75–76, 1968.