Exploring the Potential of Decision Diagrams for Efficient In-Memory Design Verification

Khushboo Qayyum
Cyber-Physical Systems, DFKI
Bremen, Germany
khushboo.qayyum@dfki.de

Abhoy Kole
Cyber-Physical Systems, DFKI
Bremen, Germany
abhoy.kole@dfki.de

Kamalika Datta
Institute of Computer Science,
University of Bremen
Cyber-Physical Systems, DFKI
Bremen, Germany
kdatta@uni-bremen.de

Muhammad Hassan
Institute of Computer Science,
University of Bremen
Cyber-Physical Systems, DFKI
Bremen, Germany
hassan@uni-bremen.de

Rolf Drechsler
Institute of Computer Science,
University of Bremen
Cyber-Physical Systems, DFKI
Bremen, Germany
drechsler@uni-bremen.de

ABSTRACT

In this paper we present the first *Decision Diagrams* (DDs) based methodology for verifying the *Resistive Random Access Memory* (ReRAM) synthesis process. In particular, we propose a methodology which leverages *Binary Decision Diagrams* (BDDs), *Multiplicative Binary Moment Diagrams* (*BMDs), and *Kronecker Multiplicative BMDs* (K*BMDs) for verification. We introduce a synthesis tool for ReRAM-compatible micro-operations and a DD generation process for equivalence checking. Experimental results on a large set of arithmetic adders demonstrate that our DD-based approach significantly outperforms SAT solvers in verification speed, offering a more efficient and scalable solution.

KEYWORDS

In-memory Computing, Verification, Word Level Decision Diagrams (WLDD), Binary Decision Diagrams (BDDs)

ACM Reference Format:
Khushboo Qayyum, Abhoy Kole, Kamalika Datta, Muhammad Hassan, and Rolf Drechsler. 2024. Exploring the Potential of Decision Diagrams for Efficient In-Memory Design Verification. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3649476.3658766

1 INTRODUCTION

The explosive growth of deep learning and massive data generation is pushing the limits of traditional computational resources. The sheer volume of data originating from sources like *Internet-of-Things* (IoT) devices and data warehouses demands efficient storage and rapid processing. As conventional computing struggles with the "Memory Wall" and "Power Wall" challenges, alternative technologies are urgently needed.

Resistive Random Access Memory (ReRAM), or memristors, have emerged as a promising solution for In-Memory Computing [7]. Their ability to perform computations offers advantages in modern data processing. Researchers have employed logic design styles (IMPLY, MAGIC, MAJ) to create ReRAM crossbar implementations and corresponding micro-operations [6, 13, 16, 17, 20, 22] termed synthesis process. Traditionally, verifying the correctness of this synthesis process relied on manual inspection.

Recently, *Boolean Satisfiability* (SAT) techniques have been used for automated equivalence checking of ReRAM logic designs [3, 8, 12, 19]. While SAT solvers are versatile, specialized techniques like *Decision Diagrams* (DDs), including *Binary DDs* (BDDs) and *Word Level DDs* (WLDDs) like *Multiplicative Binary Moment Diagrams* (*BMDs) and *Kronecker Multiplicative BMD* (K*BMDs), have proven exceptionally efficient for verifying large arithmetic adder circuits [2, 4, 5, 10, 14, 23] – a vital aspect of data processing. This established success of DDs in arithmetic circuit verification suggests their potential in the ReRAM context, a possibility yet to be fully explored.

Contribution: In this paper, we propose, for the first time a DD-based verification methodology to the best of our knowledge, to verify the In-Memory designs. In particular, we propose a methodology which leverages BDDs, *BMDs, and K*BMDs for verification. Our contribution is twofold: First, a synthesis tool is developed which transforms a *Majority-Inverter Graph* (MIG) representation of the circuit design into a micro-operations file, utilizing an intermediate form known as the *Resistive Random Access Memory Matrix* (ReMAT) [1]. This necessitates to carefully select the matrix which represents the ReRAM crossbar and consider the sequential behavior of the ReRAM crossbars. Second, we detail the generation of DDs from both the original MIG representation and the synthesized micro-operation. This step is crucial for the subsequent verification phase. By establishing an equivalence between the DD derived from the original MIG representation and the one obtained from the micro-operations, we ensure the fidelity of the synthesis process and the functional correctness of the design. Our extensive

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.
GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0605-9/24/06…$15.00
https://doi.org/10.1145/3649476.3658766

experimental evaluation on various arithmetic adders (upto 128 bits) of different underlying architectures generated using ArithsGen [15] show that DDs are 88× faster than SAT solvers in verifying a diverse class of adder circuits.

2 PROPOSED VERIFICATION METHODOLOGY

In this section, we discuss about the proposed verification methodology using various DD based methods. The proposed DD based verification methodology is shown in Figure. 1. The input to our verification tool is the MIG representation of the function. The initial MIG representation is given in the form of a *Verilog* file. The task here is to verify the micro-operation file generated using a ReRAM based synthesis tool against the initial file description provided using *Verilog* file. First, the tool takes *Verilog* file as input and generates the micro-operations file required to be executed in the crossbar. To realize this conversion, an intermediate representation known as ReMAT is used to generate the micro-operation file. The next step is to generate the clauses from the *Verilog* file and the micro-operation file. As the micro-operation file cannot be directly used to generate the clauses, the ReMAT is also used to generate the clauses as shown in Figure. 1. From these two set of clauses we generate the DDs, one for the original *Verilog* file and other for the micro-operation file. Finally the methodology checks whether the two DDs are equivalent or not. The next sub-section discusses the synthesis process to generate the micro-operation file from MIG representation.

2.1 Synthesis Process

The synthesis process comprises of two phases i.e. mirco-operation file generation and clause generation.

2.1.1 *Micro-operation File Generation.*

In this phase, the MIG representation of a circuit is converted to a ReRAM crossbar compatible micro-operation representation. To realize the crossbar functionality, the intermediate ReMAT representation is utilized. A ReMAT representation is a 2-D matrix with n rows and m columns, where n depends on the number of operations in the MIG representation and m depends on the number of input variables. Initially, n is 2 where first row contains the variables and second row contains the first operation, new rows are added dynamically to process further operations. Please note, this dynamic expansion shows the scalability of ReMAT structure to handle the full complexity of the MIG. Each operation of MIG file is analyzed for the corresponding micro-operation representation and accordingly a number of slots are reserved in the matrix. A majority operation is realized on a crossbar only when a variable on a row is complemented. Therefore, if an operand is available, that is complemented in the MIG representation, it is used directly; otherwise any operand is selected to be complemented. The operations along with their complement calculations and majority operations, are both saved in a micro-operation file and in the ReMAT structure. This dual construction ensures the correct implementation and tracking of the operations. The structure allows for the reuse of values from previous operations to optimize the process. This reuse avoids unnecessary re-initialization and leverages already computed values. This allows us to ensure that the sequential behaviour of the crossbars is preserved. Every operation in the MIG file is mapped on the ReMAT structure and subsequently its micro-operation counter-part is generated. Once all the MIG nodes are traversed in a sequential fashion, the required micro-operation file is generated that represents the original functionality of the MIG file.

2.1.2 *Clause Generation.*

In order to verify that the generation of micro-operation representation is error-free, we have to convert this representation into a format that can be processed by the subsequent DD-based tools. Therefore, using the micro-operation file and MIG file, we generate the clauses that can be used later to perform verification of the circuit. The generation of clauses from the MIG representation is relatively straightforward but clause generation from micro-operation file is complex. In the micro-operation file, multiple steps represent a single operation therefore we need to have a look-ahead mechanism to correctly generate the clauses. For the clause generation from the MIG, each operation is read and its clause is generated and saved in a file. In the case of micro-operation file, the process works in the following way. Based on the header information present in the micro-operation file, a ReMAT structure is initialized and for every step, the ReMAT structure is updated and the clauses are generated. When the ReMAT structure is used for clause generation, the size of the ReMAT remains the same as that in the header of the micro-operation file header. Using the ReMAT structure for clause generation prevents unnecessary clause to be added up, as compared to the previous works [8]. Our approach ensures that the number of clauses generated using micro-operation file are comparable to that of the number of clauses generated by MIG files. This is due to the fact that for converting the MIG file into micro-operation file and to convert the micro-operation file into clauses we use the same ReMAT structure. This structure enables to retain the previous information for clause generation.

2.2 DD Generation

While the SAT-solvers can use the generated clauses without further processing, our DD-based tool requires further processing in-order to proceed. Therefore, once the ReMAT and MIG clauses are generated, they need to be processed to extract circuit level information out of the clauses in order to generate the respective DDs. With the help of a parsing script, the circuit information is extracted and saved in a Bench format. This Bench format file is used by the DD generation tool to form a netlist of the circuit contained within the bench file. Once the netlist is ready, the DD framework starts with initializing the Principal Inputs (PI). With the PIs initialized, the construction of the DD commences one output at a time. That is, the netlist for each Principal Output (PO) is traversed one at a time, and the DD is created starting with the PIs. The process terminates when all the intermediate nodes between the PIs and POs have been traversed and the final DDs for all the POs have been completed and assigned to their respective POs. In case of the WLDDs (i.e., *BMDS and K*BMDs) an extra step of grouping of the output is performed, as opposed to the BDD based construction. Since WLDDs represent function output at word level, therefore grouping of outputs plays an important role in WLDD construction. Hence after generation of all WLDDs for individual POs, the POs are further grouped based on which bit they represent in the word-level functionality of the circuit. Once the construction is concluded and where necessary output are grouped, the DDs of the

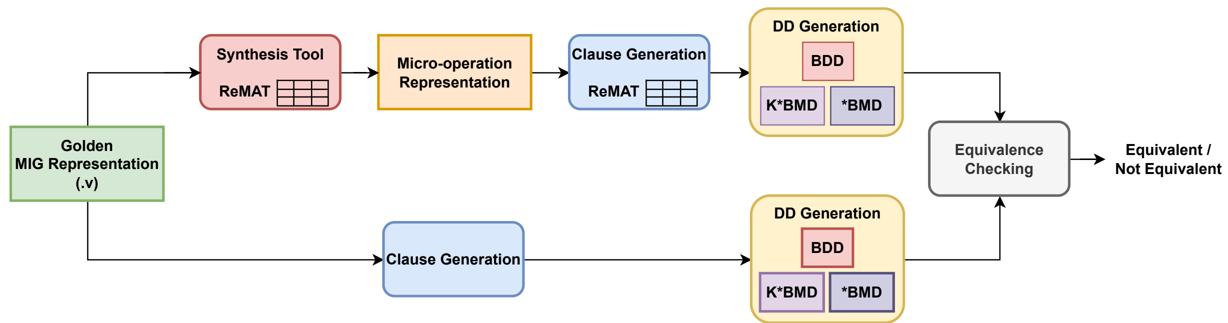

Figure 1: DD based verification methodology

circuit generated from the micro-operation representation and the MIG can be checked for equivalence.

2.3 Verification Process

To compare SAT-based methods with the DD-based methods, we perform equivalence checking using the DDs. Once the DD's are generated for the golden response (MIG) and for the design under verification in this case ReMAT, then the verification process follows. Since the DDs that we selected are also canonical in nature therefore, if the underlying functionality of two circuits is same, then the DDs generated from these circuits will remain same provided the input variable ordering used for generating the DDs are identical. This canonicity of the DDs simplifies the process of verifying the circuits and also it is an essential factor which helps in minimizing the verification effort. If the DD of a given circuit is same as the golden circuit, the functionality of the circuit is correct. In the DD packages, this verification can be performed using a pointer comparison of the DD of the given circuits with the DD of the golden circuits (in our case the MIG and ReMAT).

3 EXPERIMENTAL EVALUATION

In this section, we summarize the results of the various arithmetic adder designs for both SAT and DD based verification methods. For our experiment we generated 13 different types of adders ranging from 8 bit upto 128 bit width with the help of ArithsGen Tool [15]. For the SAT based verification, we use Z3 solver in Python environment. To perform DD based verification, we use a framework with *CUDD* and *WLDD* package at its heart. We restrict K*BMDs decomposition to only negative Davio for these experiments. The *mockturtle* library was used to convert the *Verilog* files of adders to MIG representation [21]. In Z3 solver the miter is used for the verification where outputs of the golden adder circuit and the *Design Under Verification* (DUV) are Xor'ed. To construct the BDDs, we select the interleaved variable ordering but for the WLDD based approaches we ordered the inputs in the same sequence as given in the circuit files. This is due to the fact that for WLDD's the input variable order has minimal effect [5]. All the experiments were performed on a AMD EPYC 7302P 16-Core Processor server with 512GB RAM.

3.1 Run-time Analysis

Figure 2 shows the time taken to verify all the considered 13 different types of adders of 8-bit size using all the DD based approaches (i.e., BDD, *BMD and K*BMD) and the Z3 solver. Along the y-axis the time in seconds is represented and the different types of adders are plotted in the direction of the x-axis. Please note the bottom half of the y-axis is zoomed-in to get a better insight into the performance of the BDDs and Z3 solver. It can be easily seen how BDDs perform better for almost all the different types of adders. For the CSA, all the methods show higher runtime except the BDDs which have better performance for this type of adders too. Here we also see that Z3 timings are generally up to 5× times slower than the BDD timings. As BDDs clearly outperformed among other DD based methods, we next compare the Z3 solver timings against BDDs for verification of adders of increasing size.

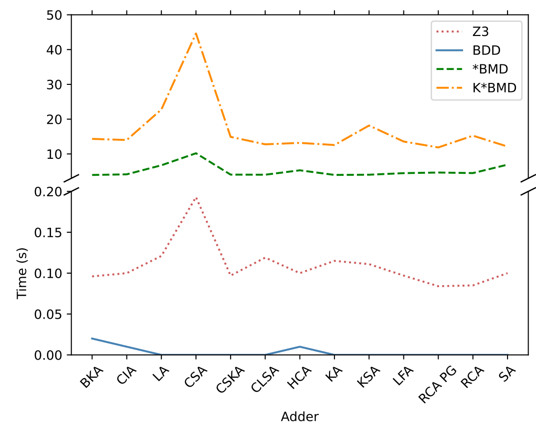

Figure 2: Timings to verify different 8-bit Adders: DDs Vs Z3

3.2 Scalability Analysis

For the comparison we have specifically selected three different adder architecture categories i.e. a simple RCA, a Carry and Propagate Adder i.e., CLA and a parallel-prefix adder i.e., KA of size 8 to 128 bit. In Figure 3, the verification time of these adders are plotted for both BDD based approach and Z3 solver. Due to experiencing higher run-time for increasing adder size, results for WLDD based approaches are omitted. While the behaviour of Z3 solver for all three adders is very similar, the BDDs perform much better with very little increase in verification time. The highest verification time for Z3 with 128 bit was for the KS at 3.3s and lowest timings were for the RCA at 1.2s. In case of BDD, with 128 bit width the largest reading was for KA at 0.1s which was still 10× smaller than the lowest Z3 value. Figure 4 shows the average time to perform for all the adders at different bit-sizes. Like the Figure 2, y-axis of

Figure 4 has been zoomed-in in the bottom half to better visualize the BDD values. The average of the BDDs at 128-bit is about 30× smaller than the values of the Z3 average. BDDs also have a better standard deviation as compared to the Z3 which means BDDs have more consistent verification timings. The standard deviation for Z3 with 128 bit is 1.53s which shows the spread of values of Z3 verification for 128-bit adders of different types.

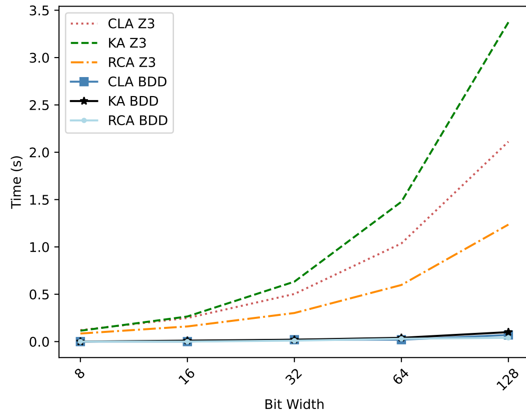

Figure 3: Time to verify: BDDs Vs Z3

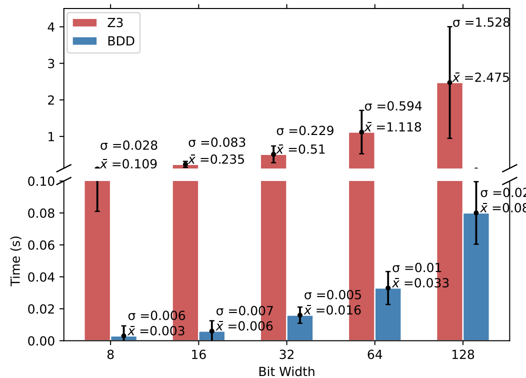

Figure 4: Average time to verify all adders: BDDs Vs Z3

3.3 Benchmarking Verification

Table 1 shows the verification results of various adders considered for evaluation. The first two columns show the type of the adders and their corresponding bit width. The third and fourth column represent the number of clauses and time taken to generate the clauses for a MIG file, respectively. Similarly the fifth and sixth column depict the clauses and the time required to generate clauses from a micro-operation file. The seventh column presents the time taken by the Z3 solver to verify the files. The eighth and ninth columns show the final number of nodes of a BDD and time required to verify the adder with the help of BDDs, respectively. The tenth column shows the final size of *BMD followed by the eleventh column shows the time taken to verify adder using *BMDs. The column twelve and column thirteen show the final node size of K*BMDs and time taken to verify the selected adders, respectively. The last column represents the improvement ratio of the best DD based method with Z3, which in our case is BDDs, w.r.t Z3.

It is evident from Table 1 that the BDD is significantly faster than the today's most advanced SAT solver for verification of arithmetic adder circuits. The construction time of BDDs are heavily dependent upon the underlying structure and when we convert the *Verilog* file to MIG structure and MIG to micro-operation file, the underlying structure of adders is vastly changed but even after that, BDDs still perform better as compared to SAT solvers[18]. We get with upto 88× faster verification time with 128 bit for CLA with the help of BDDs and smallest improvement is for RCA PG at 15×. This shows that BDD based verification is consistent in performance w.r.t. the adder structure. In Table 1, the values with * denote such a small value that we disregard the values for BDDs as they do not provide much insight into performance. For the other DDs we could only construct Adders upto 8 bit width. The package used for WLDDs is considerably old and thus less optimized as opposed to construction using Z3 and CUDD. It is important to note that the better Z3 timings in comparison to [8] can also be attributed to reduced number of clauses generated by the synthesizer with the help of the ReMAT structure. In the previous works, the number of clauses generated from micro-operation file are significantly higher than clauses generated from the MIG file with an average 2× increase [8].

4 CONCLUSION

In this paper, we presented the first DD-based methodology for verifying ReRAM synthesis process. In particular, we proposed a methodology which leverages BDDs, *BMDs and K*BMDs for verification. We developed a synthesis tool to translate MIG representations into ReRAM-compatible micro-operations and a DD generation process for equivalence checking. By generating DDs from the original MIG and synthesized micro-operations, and subsequently demonstrating their equivalence, we ensured the correctness of the synthesis process and the overall In-Memory design. Our experimental results on arithmetic adders demonstrated a significant speedup (up to 88×) when using our DD-based approach compared to SAT solvers. As a future work, we plan to extend this work towards polynomial formal verification [9, 11]. In particular, the efficiency of our methodology for polynomial upper bounds.

ACKNOWLEDGEMENT

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1 and DR 287/35-2) and in part within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1).

REFERENCES

[1] L. Amaru, P-E Gaillardon, and G. D. Micheli. 2014. Majority Inverter Graph: A novel data-structure and algorithms for efficient logic optimization. *DAC.*
[2] B. Becker, R. Drechsler, and R. Enders. 1997. On the representational power of bit-level and word-level decision diagrams. *ASP-DAC.*
[3] K. Bhunia, A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler. 2023. ReSG: A Data Structure for Verification of Majority based In-Memory Computing on ReRAM Crossbars. *TECS.*
[4] R. E. Bryant. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on.*
[5] R. E. Bryant and Y. Chen. 2001. Verification of arithmetic circuits using binary moment diagrams. *STTT.*
[6] S. Chakraborti, P.V. Chowdhary, K. Datta, and I. Sengupta. 2014. BDD based Synthesis of Boolean Functions using Memristors. *IDT.*
[7] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li. 2018. GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data. *TPDS.*
[8] A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler. 2023. Automated Equivalence Checking Method for Majority based In-Memory Computing on ReRAM Crossbars. *ASP-DAC.*

Table 1: Experimental Results of the Adders for Z3 and DDs

TYPE	BIT SIZE	MIG		MICRO OPERATION		Z3	BDD		*BMD		K*BMD		Improvement
		CLAUSES	Time(s)	CLAUSES	Time(s)	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Z3 vs BDD
BKA		88	0.08	88	0.13	0.096	39	0.02	16	3.96	16	14.33	4.81
CIA		92	0.08	92	0.13	0.195	39	0.01	16	4.14	16	14.01	10.02
CLA		111	0.09	111	0.16	0.397	39	~0	16	6.74	16	22.77	*
CSA		188	0.13	188	0.21	0.829	39	~0	16	10.2	16	44.63	*
CSKA		88	0.08	88	0.13	1.910	39	~0	16	4.06	16	14.91	*
CLSA		109	0.09	109	0.16	0.100	39	~0	16	4.03	16	12.77	*
HCA	8	91	0.08	91	0.13	0.209	39	0.01	16	5.3	16	13.18	10.01
KA		104	0.09	104	0.05	0.418	39	~0	16	3.97	16	12.57	*
KSA		100	0.08	100	0.15	0.851	39	~0	16	4.03	16	18.17	*
LFA		88	0.08	88	0.17	1.728	39	~0	16	4.48	16	13.57	*
RCA PG		76	0.07	76	0.12	0.121	39	~0	16	4.66	16	11.87	*
RCA		76	0.07	76	0.17	0.249	39	~0	16	4.5	16	15.25	*
SA		91	0.08	91	0.13	0.503	39	~0	16	6.92	16	12.13	*
BKA		189	0.12	189	0.23	1.037	79	0.01	-	T.O	-	T.O	19.47
CIA		204	0.13	204	0.21	2.112	79	~0	-	T.O	-	T.O	*
CLA		237	0.15	237	0.22	0.193	79	~0	-	T.O	-	T.O	*
CSA		481	0.18	481	0.22	0.485	79	0.02	-	T.O	-	T.O	24.23
CSKA		184	0.12	184	0.22	1.190	79	~0	-	T.O	-	T.O	*
CLSA		255	0.17	255	0.18	2.873	79	0.01	-	T.O	-	T.O	25.71
HCA	16	207	0.14	207	0.17	7.057	79	0.01	-	T.O	-	T.O	21.34
KA		254	0.15	254	0.18	0.097	79	0.01	-	T.O	-	T.O	26.55
KSA		249	0.15	249	0.18	0.188	79	~0	-	T.O	-	T.O	*
LFA		192	0.09	192	0.18	0.371	79	~0	-	T.O	-	T.O	*
RCA PG		156	0.11	156	0.09	0.742	79	0.01	-	T.O	-	T.O	15.64
RCA		156	0.11	156	0.17	1.506	79	~0	-	T.O	-	T.O	*
SA		207	0.13	207	0.18	0.119	79	0.01	-	T.O	-	T.O	21.25
BKA		400	0.18	400	0.23	0.257	159	0.01	-	T.O	-	T.O	39.69
CIA		428	0.19	428	0.23	0.546	159	0.01	-	T.O	-	T.O	41.82
CLA		489	0.17	489	0.06	1.123	159	0.02	-	T.O	-	T.O	25.14
CSA		1182	0.17	1182	0.24	2.327	159	0.02	-	T.O	-	T.O	59.49
CSKA		376	0.16	376	0.21	0.100	159	0.01	-	T.O	-	T.O	37.05
CLSA		547	0.12	547	0.17	0.213	159	0.02	-	T.O	-	T.O	27.28
HCA	32	463	0.18	463	0.17	0.464	159	0.02	-	T.O	-	T.O	23.21
KA		604	0.17	604	0.18	1.029	159	0.02	-	T.O	-	T.O	31.65
KSA		598	0.17	598	0.18	2.249	159	0.02	-	T.O	-	T.O	31.12
LFA		412	0.18	412	0.17	0.115	159	0.02	-	T.O	-	T.O	20.72
RCA PG		316	0.16	316	0.17	0.265	159	0.02	-	T.O	-	T.O	15.15
RCA		316	0.17	316	0.17	0.633	159	0.01	-	T.O	-	T.O	30.16
SA		463	0.16	463	0.17	1.477	159	0.01	-	T.O	-	T.O	46.61
BKA		849	0.16	849	0.24	3.373	319	0.04	-	T.O	-	T.O	20.74
CIA		876	0.17	876	0.23	0.111	319	0.03	-	T.O	-	T.O	28.37
CLA		993	0.17	993	0.24	0.263	319	0.02	-	T.O	-	T.O	51.84
CSA		2819	0.18	2819	0.26	0.622	319	0.04	-	T.O	-	T.O	71.82
CSKA		760	0.13	760	0.24	1.455	319	0.02	-	T.O	-	T.O	37.12
CLSA		1131	0.17	1131	0.19	3.295	319	0.02	-	T.O	-	T.O	56.14
HCA	64	1023	0.17	1023	0.17	0.097	319	0.05	-	T.O	-	T.O	20.57
KA		1402	0.17	1402	0.20	0.201	319	0.04	-	T.O	-	T.O	36.92
KSA		1395	0.18	1395	0.20	0.414	319	0.05	-	T.O	-	T.O	29.10
LFA		876	0.18	876	0.18	0.886	319	0.03	-	T.O	-	T.O	29.54
RCA PG		636	0.17	636	0.17	1.851	319	0.03	-	T.O	-	T.O	19.71
RCA		636	0.14	636	0.17	0.084	319	0.03	-	T.O	-	T.O	19.95
SA		1023	0.16	1023	0.18	0.156	319	0.03	-	T.O	-	T.O	34.55
BKA		1816	0.18	1816	0.04	0.303	639	0.1	-	T.O	-	T.O	19.10
CIA		1772	0.18	1772	0.26	0.591	639	0.1	-	T.O	-	T.O	17.28
CLA		2001	0.15	2001	0.27	1.228	639	0.07	-	T.O	-	T.O	30.17
CSA		6568	0.22	6568	0.34	0.085	639	0.08	-	T.O	-	T.O	88.21
CSKA		1528	0.17	1528	0.26	0.160	639	0.06	-	T.O	-	T.O	25.11
CLSA		2299	0.18	2299	0.22	0.302	639	0.07	-	T.O	-	T.O	33.24
HCA	128	2239	0.18	2239	0.21	0.598	639	0.11	-	T.O	-	T.O	20.45
KA		3192	0.15	3192	0.24	1.236	639	0.1	-	T.O	-	T.O	33.73
KSA		3184	0.19	3184	0.24	0.100	639	0.07	-	T.O	-	T.O	47.07
LFA		1852	0.18	1852	0.20	0.213	639	0.09	-	T.O	-	T.O	20.56
RCA PG		1276	0.20	1276	0.18	0.466	639	0.08	-	T.O	-	T.O	15.35
RCA		1276	0.17	1276	0.18	1.036	639	0.04	-	T.O	-	T.O	30.89
SA		2239	0.18	2239	0.20	2.310	639	0.07	-	T.O	-	T.O	33.00

BKA = Brent Kung Adder; CIA= Carry Increment Adder; CLA= Carry Lookahead Adder; CSA = Conditional Sum Adder
CSKA = Carry Skip Adder; CLSA = Carry Select Adder; HCA = Hans Carlson Adder; KA = Knowles Adder
KSA = Kogge Stone Adder; LFA = Ladner Fischer Adder; RCA PG = Ripple Carry adder with Propagate / Generate
RCA = Ripple Carry Adder; SA = Sklansky Adder

[9] R. Drechsler. 2021. PolyAdd: Polynomial Formal Verification of Adder Circuits. *DDECS*.

[10] R. Drechsler, B. Becker, and S. Ruppertz. 1997. The K*BMD: A verification data structure. *IEEE Design & Test of Computers*.

[11] R. Drechsler and A. Mahzoon. 2022. Polynomial Formal Verification: Ensuring Correctness under Resource Constraints : (Invited Paper). *ICCAD*.

[12] S. Froehlich and R. Drechsler. 2022. Generation of Verified Programs for In-Memory Computing. *DSD*.

[13] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta. 2018. A Scalable In-Memory Logic Synthesis Approach Using Memristor Crossbar. *TVLSI*.

[14] S. Höreth and R. Drechsler. 1999. Formal verification of word-level specifications. *DATE*.

[15] J. Klhufek and V. Mrazek. 2022. ArithsGen: Arithmetic Circuit Generator for Hardware Accelerators. *DDECS*.

[16] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. 2014. MAGIC—Memristor-aided logic. *TCAS-II*.

[17] F. Lalchhandama, M. Sahani, V. M. Srinivas, I. Sengupta, and K. Datta. 2022. In-Memory Computing on Resistive RAM Systems Using Majority Operation. *JCSC*.

[18] K. Qayyum, A. Mahzoon, and R. Drechsler. 2022. Monitoring the Effects of Static Variable Orders on the Construction of BDDs. *MESIICON*.

[19] F. Shirinzadeh, A. Deb, S. Shirinzadeh, A. Kole, K. Datta, and R. Drechsler. 2024. In-Memory SAT-Solver for Self-Verification of Programmable Memristive Architectures. *VLSID*.

[20] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler. 2016. Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs. *DATE*.

[21] M. Soeken et al. 2018. The EPFL logic synthesis libraries. *arXiv preprint arXiv:1805.05121*.

[22] P. L. Thangkhiew, R. Gharpinde, and K. Datta. 2018. Efficient mapping of Boolean functions to memristor crossbar using MAGIC NOR gates. *TCAS-I*.

[23] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. 2009. SWORD: A SAT like prover using word level information. *VLSI-SoC*.