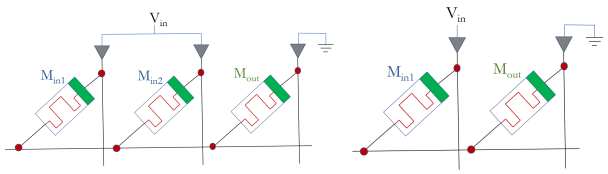
Benchmarking Multiplier Architectures for MAGIC Based In-Memory Computing

Chandan Kumar Jha[Ш] Rolf Drechsler[Ш,†]
Department of Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany[Ш]
Department of Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany[†]
chajha@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—A wide variety of multiplier architectures optimized for area, delay, and energy have been proposed in the literature. These multipliers have been extensively studied for CMOS technology. While in-memory computing (IMC) using memristors has garnered significant interest in recent years, multiplier designs have received far less attention. In this work, we aim to bridge this gap and for the first time analyze and compare diverse multiplier architectures for IMC based on memristors. We analyze 275 different signed and unsigned multiplier architectures for the 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit multipliers. We have used the state-of-the-art mapping tool called SIMPLER to perform the mapping of these multiplier designs to the memristor crossbars. We have used the memristor count and the number of cycles as design metrics to identify the most suitable architectures for IMC using the MAGIC design style. We show that there is a difference of 1.2× in the design metrics between the best and worst multiplier architectures across all bit widths. We also show that several Array and Dadda Tree based multipliers are best suited for 4x4 multipliers. Multipliers having Dadda Tree based partial product accumulator and Serial Prefix final stage adder are best suited for 8-bit or higher bit width multipliers. We will make all the multiplier designs and memristor crossbar mapping files generated from SIMPLER open-source at https://github.com/agra-uni-bremen/newcas2023-magic-multiplier-lib. We believe that our work will act as a benchmark for future works in this direction and the designers can use them to perform further optimizations, synthesis for other design styles, verification, etc.**

I. Introduction

In-memory computing (IMC) has attracted the interest of both industry and academia in recent years. IMC aims to reduce the issue of memory bottleneck in the conventional von Neumann architectures [1]. It also improves the energy efficiency as it reduces the to and fro data movement between memory and processing/compute unit [2]. IMC using memristors has been extensively explored as memristors can act both as storage and compute units [3]. A memristor is a two-terminal device that can change its resistance depending upon the magnitude and direction of the applied voltage across its terminals [4]. It can be used to perform both analog and digital computations [5], [6]. In this work, we focus on using memristors to perform digital computations [7]. Memristors can be configured to be in a high resistance state (HRS), i.e., logic 0 or a low resistance state (LRS), i.e., logic 1. These states can then be used to perform digital computations. There are several design styles to perform logic computation using memristors [8]–[14]. In this work, we have used one of the most popular design styles called the MAGIC design style [13]. The NOR and NOT operations performed using this design style can be mapped to memristor crossbars and hence are suitable for IMC [13].



(a) NOR (b) NOT

Fig. 1. MAGIC NOR and NOT Gates

Multiplication is one of the most common operations that need to be performed in applications [15]. Handcrafted in-memory multipliers designs based on memristors have received interest in recent years and remain an active area of research [16]–[18]. However, the analysis is limited to a single architecture or for a subset of bit widths. Hence, there is a need for overall analysis and comparison between different architectures and bit widths. Multiplication can be performed using a wide variety of architectures [19]–[21]. Each of these architectures has been studied in detail for digital IC designs [15], however, the same is not true for multipliers based on memristors for IMC. In this work, for the first time, we investigate these architectures for IMC using memristors. Since these architectures are optimized for digital IC design, the same designs when mapped to memristors can have different design properties. This makes it necessary to perform analysis to identify the best multiplier architectures when they are mapped to memristor crossbars using a design style. Our work has the following contributions:

- In this work, we present the first in-depth analysis and comparison of various multipliers implemented using the MAGIC design style on memristor crossbars.
- We analyzed 55 different signed and unsigned multipliers designs for each of the 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit multipliers, i.e., a total of 275 multiplier designs.
- We used the number of gates and the total cycles as metrics for the evaluation of the multiplier designs mapped to memristor crossbars.
- We show the difference between the best and worst case multiplier designs is 1.2× across all bit widths.
- We also show that Dadda Tree based partial product accumulator and Serial Prefix adder are best suited for

TABLE I
VARIOUS MULTIPLIER ARCHITECTURES

Bit Width	Partial Product Generator	Partial Product Accumulation	Final Stage Adder
4x4	Signed (**S**)	Array (**AR**)	Ripple Carry (**RC**)
8x8	Unsigned (**U**)	Counter-Based Wallace Tree (**CWT**)	Carry Look Ahead Adder (**CLA**)
16x16		Wallace Tree (**WT**)	Lander-Fischer (**LF**)
32x32		Dadda Tree (**DT**)	Kogge-Stone (**KS**)
64x64			Brent-Kung (**BK**)
			Carry Skip (**CS**)
			Serial Prefix Adder (**SE**)

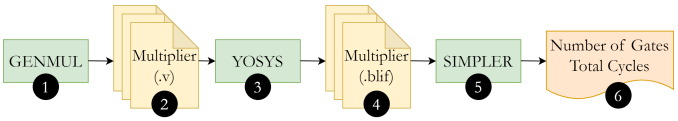


Fig. 2. Framework used for Analysis

implementing multipliers using MAGIC design style to perform IMC using memristors.

The rest of the paper is organized as follows. In Section II, we discuss the necessary background. In Section III, we discuss the framework used for performing the analysis. In Section IV, we discuss the comparison results of various multiplier designs and in Section V, we conclude the paper.

II. BACKGROUND

In this section, we discuss the necessary background of the MAGIC gates [13], the mapping tool SIMPLER [22], and multiplier architectures.

A. MAGIC Gates

In this work, we have used the MAGIC design style to implement the logic function using memristors [13]. The NOR and the NOT function can be mapped to the crossbar array as shown in Fig. 1. The output memristor (M_{out}), is set to 1 before the evaluation of any operation. Since it is a two-input NOR gate the total number of input combinations is 4. For all combinations except 00, there exists a path between the V_{in} and ground. Hence the current flows through the output memristor, increasing its resistance. This changes the logic state of the output memristor from 1 to 0. When the input is 00, the output memristor remains in the low resistance state maintaining its logic state 1. For the NOT operation the current flows when the input memristor is in a low resistance state, i.e., logic 1, changing the logic state of the output memristor to 0. For the case when the input memristor is in a high resistance state the output memristor maintains logic 1.

B. Multiplier Design

Several methods have been developed over the years to design multipliers targeting area, power, and delay but the overall architecture remains the same [23]. The first stage consists of the partial product generator. This generates the partial products depending upon the inputs to the multiplier. The next stage is the partial product accumulator which adds all the partial products to generate an intermediate output. The intermediate output is fed to the final stage adder to finally obtain the multiplication output. There are several ways to implement the partial product accumulator and the final stage adder as shown in Table I. In this work, we have used the GENMUL tool to generate the multiplier designs [23].

III. EVALUATION FRAMEWORK

The overall evaluation framework is shown in Fig. 2. We will now discuss the stages of the framework in detail.

A. Multiplier Design Generation (GENMUL)

In the first stage, we used GENMUL to generate the multiplier designs as shown in Fig. 2 ❶. GENMUL is an open source multiplier generator that takes the following parameters as input: a) bit width, b) partial product generator type, c) partial product accumulator type, and d) final stage adder type [23]. The values of these parameters are shown in Table I. We have used GENMUL to generate multipliers ranging from bit width of 4 to 64 for both signed and unsigned multipliers as shown in Fig. 2 ❷. We also generated all possible combinations of partial product generators and partial product accumulators[1]. Thus we have 28 and 27 unsigned and signed multiplier designs respectively for each bit width. Hence, overall we analyze 275 different multiplier designs.

B. Intermediate Synthesis (Yosys)

We have synthesized the Verilog design generated from GENMUL using Yosys to generate the Berkeley Library Exchange Format (.blif) as shown in Fig. 2 ❸. We used the *synth* and *flatten* commands of Yosys to generate the .blif files of the multiplier designs as shown in Fig. 2 ❹ [24]. This was done to make the designs compatible with the next stage of mapping them to memristor crossbars.

C. Mapping to Crossbar (SIMPLER)

SIMPLER is the state-of-the-art tool used to map the design to a memristor crossbar as shown in Fig. 2 ❺. In the final stage, we passed the .blif multiplier designs to the SIMPLER tool. SIMPLER performs the mapping of these designs to a memristor crossbar array using MAGIC NOR and NOT gates. SIMPLER maps these designs to a single row and reuses the memristors when necessary. There are various knobs for optimization in SIMPLER. We found the minimum number of memristors in a row that can be used to implement multiplier designs. We started with 25 memristors and kept increasing in steps of 25 until all the designs for a particular bit width had a mapping using SIMPLER. The number of gates and the total cycles were then obtained as metrics for evaluating the design as shown in Fig. 2 ❻.

IV. RESULTS AND DISCUSSION

In this section, we discuss the results obtained using our framework. Table II to Table VI, shows the result from 4-bit to 64-bit multipliers. The design names are abbreviated in the tables. For example, Dadda Tree partial product accumulator with Serial Prefix Adder as *DT_SE* as also shown in Table I.

[1]GENMUL throws an exception for a signed array multiplier with carry skip adder. Hence we have one less signed multiplier.

TABLE II
4X4 MULTIPLIER DESIGNS WITH 50 MEMRISTORS

Unsigned			Signed		
Designs	**Number of Gates**	**Total Cycles**	**Designs**	**Number of Gates**	**Total Cycles**
CWT_CL	154	159	**CWT_KS**	151	156
WT_LF	140	144	**DT_CL**	142	146
WT_SE	140	144	**CWT_RC**	150	154
AR_CK	130	134	**WT_RC**	150	154
DT_LF	**128**	**132**	**DT_CK**	141	145
WT_CL	154	159	**WT_CK**	140	144
DT_KS	132	136	**DT_LF**	134	138
AR_CL	**128**	**132**	**WT_CL**	161	166
DT_BK	**128**	**132**	**AR_KS**	**131**	**135**
DT_SE	**128**	**132**	**AR_LF**	**131**	**135**
AR_BK	**128**	**132**	**CWT_BK**	151	156
DT_RC	133	137	**CWT_SE**	133	137
CWT_KS	140	144	**AR_RC**	**131**	**135**
CWT_RC	143	147	**DT_SE**	134	138
CWT_CK	140	144	**CWT_CL**	161	166
CWT_SE	140	144	**AR_CL**	**131**	**135**
WT_KS	140	144	**AR_SE**	**131**	**135**
CWT_LF	140	144	**WT_BK**	151	156
WT_BK	140	144	**AR_BK**	**131**	**135**
AR_RC	130	134	**DT_BK**	134	138
WT_CK	140	144	**CWT_LF**	151	156
DT_CL	141	146	**WT_KS**	151	156
DT_CK	135	139	**WT_LF**	151	156
AR_KS	**128**	**132**	**DT_KS**	140	144
CWT_BK	140	144	**DT_RC**	142	146
WT_RC	143	147	**CWT_CK**	140	144
AR_SE	**128**	**132**	**WT_SE**	133	137
AR_LF	**128**	**132**			

TABLE III
8X8 MULTIPLIER DESIGNS WITH 75 MEMRISTORS

Unsigned			Signed		
Designs	**Number of Gates**	**Total Cycles**	**Designs**	**Number of Gates**	**Total Cycles**
CWT_CL	774	808	**CWT_RC**	712	738
WT_SE	693	721	**WT_LF**	745	780
CWT_SE	724	751	**WT_SE**	709	737
AR_CK	665	690	**WT_RC**	704	731
CWT_RC	703	730	**DT_CL**	714	743
DT_LF	690	719	**CWT_LF**	756	785
CWT_KS	774	808	**WT_BK**	745	780
WT_RC	691	719	**AR_LF**	677	705
DT_RC	664	689	**CWT_CL**	777	809
DT_KS	709	739	**DT_BK**	678	704
AR_CL	666	691	**AR_CL**	677	705
AR_KS	665	690	**CWT_KS**	777	809
AR_LF	664	689	**DT_CK**	653	677
WT_CL	780	815	**AR_KS**	677	705
DT_CK	658	683	**AR_RC**	677	705
DT_CL	726	764	**AR_BK**	677	705
WT_CK	691	719	**WT_KS**	781	820
CWT_BK	730	758	**DT_KS**	716	744
WT_LF	725	758	**AR_SE**	677	705
AR_SE	663	688	**WT_CL**	820	857
AR_BK	663	688	**DT_LF**	685	711
DT_BK	683	711	**WT_CK**	708	736
AR_RC	672	698	**CWT_BK**	756	785
WT_BK	725	758	**DT_RC**	666	690
WT_KS	762	799	**CWT_SE**	732	759
DT_SE	**648**	**671**	**DT_SE**	**651**	**673**
CWT_LF	730	758	**CWT_CK**	716	742
CWT_CK	712	739			

A. 4x4 Multipliers

The result for the 4x4 multipliers is shown in Table II. The minimum number of memristors required for mapping is 50. For unsigned multipliers, the number of gates and the cycle count range from 128-154 and 132-159 respectively. For signed multipliers, the number of gates and the cycle count range from 131-161 and 135-166 respectively. The best designs in terms of cycles are the ones based on Array and Dadda Tree based partial product accumulation. The best designs use 128 gates and the number of cycles required for multiplication is 132 as highlighted in Table II.

B. 8x8 Multipliers

The result for the 8x8 multipliers is shown in Table III. The minimum number of memristors required for mapping is 75. For unsigned multipliers, the number of gates and the cycle count range from 648-780 and 671-815 respectively. For signed multipliers, the number of gates and the cycle count range from 651-820 and 673-857 respectively. The best designs in terms of cycles are the ones based on Dadda Tree based partial product accumulation and Serial Prefix adders. The best unsigned multiplier has 648 gates and the number of cycles required for multiplication is 671 as highlighted in Table III. The best signed multiplier has 651 gates and the number of cycles required for multiplication is 673 as highlighted in Table III.

TABLE IV
16X16 MULTIPLIER DESIGNS WITH 175 MEMRISTORS

Unsigned			Signed		
Designs	**Number of Gates**	**Total Cycles**	**Designs**	**Number of Gates**	**Total Cycles**
DT_CK	2848	2883	**DT_BK**	2935	2973
WT_KS	3369	3423	**AR_BK**	3021	3065
AR_SE	2983	3025	**WT_CL**	3386	3444
CWT_RC	3147	3188	**WT_KS**	3377	3431
AR_BK	2985	3027	**CWT_LF**	3384	3430
CWT_CL	3469	3518	**AR_LF**	3021	3065
AR_CK	2993	3036	**CWT_KS**	3500	3552
AR_KS	2989	3032	**WT_RC**	3074	3113
WT_SE	3069	3108	**CWT_SE**	3243	3286
DT_KS	3090	3134	**WT_LF**	3210	3254
DT_CL	3119	3163	**AR_RC**	3021	3065
DT_SE	**2822**	**2856**	**DT_RC**	2858	2893
AR_LF	2987	3029	**CWT_RC**	3195	3237
AR_RC	3002	3045	**DT_CK**	2841	2876
CWT_KS	3489	3541	**AR_CL**	3021	3065
CWT_LF	3352	3397	**AR_KS**	3021	3065
DT_BK	2931	2970	**WT_BK**	3171	3214
DT_LF	2947	2986	**CWT_CL**	3506	3556
CWT_SE	3207	3249	**DT_KS**	3094	3137
WT_RC	3064	3103	**DT_LF**	2954	2992
WT_CL	3409	3470	**DT_SE**	**2825**	**2859**
CWT_CK	3213	3255	**AR_SE**	3021	3065
WT_LF	3198	3241	**DT_CL**	3114	3161
WT_CK	3069	3108	**CWT_CK**	3315	3359
DT_RC	2857	2892	**WT_CK**	3073	3112
AR_CL	2976	3018	**CWT_BK**	3353	3399
CWT_BK	3311	3356	**WT_SE**	3078	3117
WT_BK	3160	3203			

C. 16x16 Multipliers

The result for the 16x16 multipliers is shown in Table IV. The minimum number of memristors required for mapping is 175. For unsigned multipliers, the number of gates and the cycle count range from 2822-3489 and 2856-3541 respectively. For signed multipliers, the number of gates and the cycle count range from 2825-3506 and 2859-3556 respectively. The best designs in terms of cycles are the ones based on Dadda Tree based partial product accumulation and Serial Prefix adders. The best unsigned multiplier has 2822 gates and the number of cycles required for multiplication is 2856 as highlighted in Table IV. The best signed multiplier has 2825 gates and the number of cycles required for multiplication is 2859 as highlighted in Table IV.

D. 32x32 Multipliers

The result for the 32x32 multipliers is shown in Table V. The minimum number of memristors required for mapping is 350. For unsigned multipliers, the number of gates and the cycle count range from 11716-14452 and 11792-14575. For signed multipliers, the number of gates and the cycle count range from 11729-14486 and 11804-14602 respectively. The best designs in terms of cycles are the ones based on

TABLE V
32X32 MULTIPLIER DESIGNS WITH 350 MEMRISTORS

	Unsigned			Signed	
Designs	**Number of Gates**	**Total Cycles**	**Designs**	**Number of Gates**	**Total Cycles**
CWT_BK	13682	13773	**WT_BK**	12910	12999
WT_BK	12909	12999	**AR_RC**	12701	12799
DT_KS	12570	12665	**DT_CK**	11839	11915
WT_RC	12596	12677	**DT_KS**	12590	12685
DT_BK	12031	12113	**AR_CL**	12701	12799
CWT_LF	13782	13874	**CWT_LF**	13858	13950
WT_KS	13624	13741	**CWT_CK**	13545	13634
DT_CK	11822	11899	**AR_BK**	12701	12799
AR_KS	12614	12710	**CWT_CL**	14486	14602
WT_LF	13100	13192	**WT_CK**	12776	12861
CWT_CK	13494	13583	**WT_LF**	13091	13182
AR_RC	12653	12749	**WT_KS**	13613	13727
CWT_SE	13336	13422	**DT_RC**	11812	11888
DT_SE	**11716**	**11792**	**WT_RC**	12579	12661
AR_CL	12695	12793	**AR_SE**	12701	12799
DT_LF	12109	12191	**CWT_BK**	13714	13805
DT_RC	11797	11874	**DT_SE**	**11729**	**11804**
WT_SE	12604	12686	**AR_KS**	12701	12799
AR_CK	12632	12728	**CWT_RC**	13324	13410
AR_SE	12615	12711	**WT_CL**	13842	14022
AR_LF	12632	12728	**CWT_KS**	14312	14417
WT_CL	13771	13935	**CWT_SE**	13390	13476
WT_CK	12780	12865	**DT_BK**	12037	12118
CWT_RC	13310	13396	**DT_CL**	12877	12987
CWT_KS	14248	14353	**AR_LF**	12701	12799
AR_BK	12625	12721	**WT_SE**	12611	12693
CWT_CL	14452	14575	**DT_LF**	12129	12210
DT_CL	12761	12867			

TABLE VI
64X64 MULTIPLIER DESIGNS WITH 700 MEMRISTORS

	Unsigned			Signed	
Designs	**Number of Gates**	**Total Cycles**	**Designs**	**Number of Gates**	**Total Cycles**
WT_BK	51251	51421	**DT_RC**	48064	48219
DT_BK	48808	48972	**AR_RC**	52029	52234
DT_CK	48217	48375	**WT_KS**	53251	53469
DT_KS	50565	50763	**AR_LF**	52029	52234
DT_SE	**47892**	**48047**	**WT_BK**	51264	51433
DT_CL	51645	51890	**WT_SE**	50456	50617
CWT_BK	55539	55721	**AR_KS**	52029	52234
CWT_KS	57082	57294	**WT_LF**	51772	51944
CWT_CK	54651	54828	**DT_KS**	50338	50528
WT_KS	53224	53441	**CWT_RC**	54276	54450
WT_SE	50444	50605	**DT_CL**	51728	51990
CWT_RC	54207	54380	**AR_SE**	52029	52234
DT_RC	48067	48222	**DT_SE**	**47898**	**48052**
AR_CL	52632	52842	**AR_CL**	52029	52234
CWT_SE	54271	54444	**DT_CK**	48216	48373
AR_RC	51921	52124	**CWT_CK**	54727	54905
AR_LF	51814	52016	**AR_BK**	52029	52234
WT_CK	50930	51095	**CWT_CL**	57975	58232
CWT_LF	56126	56310	**WT_RC**	50399	50560
DT_LF	49173	49339	**CWT_KS**	57272	57485
AR_KS	51866	52069	**CWT_SE**	54336	54510
WT_RC	50393	50554	**CWT_LF**	56081	56266
CWT_CL	58223	58490	**DT_BK**	48793	48956
AR_BK	51873	52076	**CWT_BK**	55612	55795
WT_CL	54166	54506	**WT_CK**	50951	51117
AR_SE	51847	52050	**DT_LF**	49132	49296
WT_LF	51811	51984	**WT_CL**	54110	54457
AR_CK	51969	52173			

Dadda Tree based partial product accumulation and Serial Prefix adders. The best unsigned multiplier has 11716 gates and the number of cycles required for multiplication is 11792 as highlighted in Table V. The best signed multiplier has 11729 gates and the number of cycles required for multiplication is 11804 as highlighted in Table V.

E. 64x64 Multipliers

The result for the 64x64 multipliers is shown in Table VI. The minimum number of memristors required for mapping is 700. For unsigned multipliers, the number of gates and the cycle count range from 47892-58223 and 48047-58490 respectively. For signed multipliers, the number of gates and the cycle count range from 47898-57975 and 48052-58232 respectively. The best designs in terms of cycles are the ones based on Dadda Tree based partial product accumulation and Serial Prefix adders. The best unsigned multiplier has 47892 gates and the number of cycles required for multiplication is 48047 as highlighted in Table VI. The best signed multiplier has 47898 gates and the number of cycles required for multiplication is 48052 as highlighted in Table VI.

F. Overall Analysis (Inter Bit Width)

Overall we see that Dadda Tree with Serial Prefix adder requires the least number of gates and cycles for all bit widths. For 4x4 bit width, both Array multiplier and Dadda Tree multipliers give the best mapping. We see that as we move from bit width 4 to 8, 16, 32, and 64 the minimum number of memristors required for mapping increases by 1.5×, 3.5×, 7×, and 14× as compared to 4-bit respectively. For the best case gate count and the number of cycles as we move from bit width 4 to 8, 16, 32, and 64 increase by around 5×, 22×, 91×, and 374× respectively as compared to the 4-bit multiplier.

G. Overall Analysis (Intra Bit Width)

We now look at the overall variation in the metrics for a given bit width. We see that the difference between the worst case and the best case multiplier designs is around 1.2× for both gate count and the number of cycles for all the bit widths. Hence, by performing the design space exploration and comparisons across different multiplier architectures we can gain benefits of 1.2×, as compared to arbitrarily choosing any multiplier architecture. The best multiplier architecture can then be further optimized to gain more benefits. Since there are several design styles using memristors, we believe that this sort of analysis is necessary and useful as it helps us to identify the best multiplier architecture for a given memristor based design style. Since the designs and the mapping output of the SIMPLER tool will be made open-source, these designs can not only be used as benchmarks but can also be further optimized, mapped to other design styles, and used as inputs for verification methodologies tailored for memristors.

V. CONCLUSION

In this work, we present an in-depth analysis of 275 different signed and unsigned multiplier architectures for bit width ranging from 4 to 64 for IMC. We used the state-of-the-art mapping tool and performed a thorough comparison to obtain the number of gates and the total cycles across different architectures. We observed that across different multiplier architectures, there is a 1.2× difference in these design metrics making this sort of study very important. We also identified that the design that uses Dadda Tree based partial product accumulator and Serial Prefix adder is the best suited for MAGIC-based IMC. Array based partial product accumulation also gives the best designs for a bit width of 4. We will make the multiplier designs and mapping obtained using SIMPLER open-source to facilitate further research in this direction.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems*. Springer, 2023, pp. 171–243.

[2] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[3] M. Di Ventra, Y. V. Pershin, and L. O. Chua, "Circuit elements with memory: memristors, memcapacitors, and meminductors," *Proceedings of the IEEE*, vol. 97, no. 10, pp. 1717–1724, 2009.

[4] D. Strukov, G.S.Snider, D. Stewart, and R. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[5] Y. Yang, J. Joshua, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[6] N. Xu, T. Park, K. J. Yoon, and C. S. Hwang, "In-memory stateful logic computing using memristors: Gate, calculation, and application," *physica status solidi (RRL)–Rapid Research Letters*, vol. 15, no. 9, p. 2100208, 2021.

[7] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P.-E. Gaillardon, and S. Kvatinsky, "Memristive logic: A framework for evaluation and comparison," in *27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2017, pp. 1–8.

[8] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Mrl—memristor ratioed logic," in *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*. IEEE, 2012, pp. 1–6.

[9] J. Rajendran, H. Manem, R. Karri, and G. S. Rose, "An energy-efficient memristive threshold logic circuit," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 474–487, 2012.

[10] G. Snider, "Computing with hysteretic resistor crossbars," *Applied Physics A*, vol. 80, no. 6, pp. 1165–1172, 2005.

[11] L. Guckert and E. E. Swartzlander, "Mad gates—memristor logic design using driver circuitry," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 2, pp. 171–175, 2016.

[12] L. Xie, H. A. Du Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Fast boolean logic mapped on memristor crossbar," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015, pp. 335–342.

[13] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "(MAGIC) - Memristor-Aided Logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, Nov 2014.

[14] J. Borghetti, G. Snider, P. Kuekes, J. Yang, D. Stewart, and R. Williams, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[15] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*. Prentice hall Englewood Cliffs, 2002, vol. 2.

[16] S. Muthulakshmi, C. S. Dash, and S. Prabaharan, "Memristor augmented approximate adders and subtractors for image processing applications: An approach," *AEU-International Journal of Electronics and Communications*, vol. 91, pp. 91–102, 2018.

[17] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2013, pp. 242–247.

[18] V. Lakshmi, J. Reuben, and V. Pudi, "A novel in-memory wallace tree multiplier architecture using majority logic," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 3, pp. 1148–1158, 2021.

[19] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on computers*, vol. 100, no. 12, pp. 1045–1047, 1973.

[20] L. Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, pp. 349–356, 1965.

[21] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on electronic Computers*, no. 1, pp. 14–17, 1964.

[22] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2019.

[23] A. Mahzoon, D. Große, and R. Drechsler, "Genmul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*. Springer, 2021, pp. 177–191.

[24] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.