ApprOchs: A Memristor-Based In-Memory Adaptive Approximate Adder

Dominik Ochs*, Lukas Rapp*, Leandro Borzyk*, Nima Amirafshar, and Nima TaheriNejad

***Abstract*—As silicon scaling nears its limits and the *Big Data* era unfolds, in-memory computing is increasingly important for overcoming the *Von Neumann* bottleneck and thus enhancing modern computing performance. One of the rising in-memory technologies are *Memristors*, which are resistors capable of memorizing state based on an applied voltage, making them useful for storage and computation. Another emerging computing paradigm is *Approximate Computing*, which allows for errors in calculations to in turn reduce die area, processing time and energy consumption. In an attempt to combine both concepts and leverage their benefits, we propose the memristor-based adaptive approximate adder *ApprOchs* - which is able to selectively compute segments of an addition either approximately or exactly. ApprOchs is designed to adapt to the input data given and thus only compute as much as is needed, a quality current State-of-the-Art (SoA) in-memory adders lack. Despite also using OR-based approximation in the lower k bit, ApprOchs has the edge over S-SINC because ApprOchs can skip the computation of the upper n-k bit for a small number of possible input combinations (22k of 22n possible combinations skip the upper bits).**

Compared to SoA in-memory approximate adders, ApprOchs outperforms them in terms of energy consumption while being highly competitive in terms of error behavior, with moderate speed and area efficiency. In application use cases, ApprOchs demonstrates its energy efficiency, particularly in machine learning applications. In MNIST classification using Deep Convolutional Neural Networks, we achieve 78.4% energy savings compared to SoA approximate adders with the same accuracy as exact adders at 98.9%, while for k-means clustering, we observed a 69% reduction in energy consumption with no quality drop in clustering results compared to the exact computation. For image blurring, we achieve up to 32.7% energy reduction over the exact computation and in its most promising configuration ($k = 3$), the ApprOchs adder consumes 13.4% less energy than the most energy-efficient competing SoA design (S-SINC+), while achieving a similarly excellent median image quality at 43.74dB PSNR and 0.995 SSIM.

***Index Terms*—Memristive Computing, Approximate Computing, In-Memory Computing, Adaptive Adder**

I. Introduction

NOWADAYS, because of the increase in the quantity of data, processing systems are required to move ever more data rather than processing it. However, data retrieval occupies more time than its processing. Hence, research focus has been shifting towards in-memory computing. In-memory computing describes the execution of operations such as additions, multiplications or bitwise operations directly in the memory. That prevents costly data movement, saving time and energy. A subparadigm of this technology is based on memristors, which are resistors capable of memorizing states based on an applied voltage [1]. They can also be controlled to perform logical operations, such as implications (IMPLY Logic [2]) or OR-operations (FELIX Logic [3]), making them not only interesting for storage [4], but also for computation [2].

Another paradigm is approximate computing. Numerous particular applications do not require exact computing. Many applications are inherently error-tolerant, such as machine learning, image processing, and signal processing [5]–[7]. Therefore, approximate computing allows to achieve high efficiency in speed, area, and power or energy consumption by compromising accuracy [8]. Approximate computing can be coupled with memristive in-memory computing to reduce energy consumption even further [9], [10].

Improvements of memristive adders in terms of energy consumption, runtime and accuracy are of particular interest because additions are used ubiquitously and therefore carry a huge potential for optimization [11]. To the best of our knowledge, all memristive adders are run-time static, meaning a pre-defined segment of bits is approximated [12]–[15]. Depending on the application and the corresponding value distribution, that might not be suitable. Oftentimes, the lower bits of a number are approximated because they contribute less to the number. However, if small numbers often occur in an application, approximation introduces large relative errors. Furthermore, when numbers are sufficiently small, the upper bits are zero and need no computation. For CMOS, such an adaptive approximate adder has shown promising results compared to static alternatives [16]. However, there has not been a proposal for an in-memory counterpart.

To address these challenges, our key contributions are:

- We introduce the Adaptive Precision Adder Framework (APAF), an in-memory framework that splits a number into segments with a fixed size at design time, and then dynamically determines which segments of a number to compute exactly, approximately, or not at all based on the bit pattern of the input segments (at runtime) to improve speed, efficiency, and accuracy.
- The APAF allows for any exact or approximate adder to be used within its framework. This enables the user to tailor the involved exact and approximate adders to their needs, acting as a meta optimization for any previously proposed approximate adder.
- We evaluate the APAF on an established exact in-memory adder paired with a novel approximate adder, which ORs

Authors are with the Institute of Computer Engineering, Heidelberg University, 69117 Heidelberg, Germany, (e-mail: {dominik.ochs, lukas.rapp, leandro.borzyk} @stud.uni-heidelberg.de, {nima.amirafshar, nima.taherinejad}@uni-heidelberg.de)

* The authors have contributed equally to this work.

the inputs, as a proof of concept. We call this specific adder and instantiation of the APAF, the ApprOchs adder.

This adaptive approach aims to optimize the accuracy and efficiency of addition operations across various applications while allowing for flexibility and adjustability.

The rest of this paper is structured as follows. We begin with an introduction into the ApprOchs adders architecture in Section III, followed by a look at the circuit implementation. The corresponding costs are presented in Section IV. After that, we examine machine learning applications in Section VI and image processing applications in Section VII employing the ApprOchs adder, before finally drawing a conclusion and taking an outlook on future development in Section VIII.

II. Related Work

A. Memristors and Memristor Logic

The memristor, a recently developed two-terminal device, was first theorized by Chua [1] and later experimentally demonstrated by Strukov et al. [17]. This component functions as a non-volatile memory element by retaining information through its variable resistive states. Due to its benefits, including minimal power consumption, rapid data writing capabilities, and compact size, the memristor has emerged as a highly promising candidate for memory cell applications [2], [12], [18], [19].

IMPLY represents a type of stateful logic that takes advantage of the inherent properties of memristors, enabling logical functions to be carried out without the necessity of distinct read or write steps [19]. This logic is highly compatible with crossbar array architectures and has emerged as a leading approach for in-memory computing systems [18], [20]. In an IMPLY operation, typically expressed as $a =) b$, the input values are determined by the resistive states of the memristors, with the corresponding truth table displayed in Table I.

FELIX is another type of stateful logic, which also enables logical functions without distinct read or write steps [3]. The FELIX logic, or more specifically the FELIX-OR, allows a single-cyle logical OR operation denoted by $a \_ b$ [3]. Table I contains the logic table. Additionally, there are other stateful logics such as SIXOR [19], MAGIC [21], and TSML [22]. For our exact adder, we used IMPLY because it is the most reliable in memristive technology [14]. However, our approach can be expanded to use exact and approximate adders from other logics and architectures. For our approximate-part adder and adaption process, we used FELIX because it enables us to model an OR in a single cycle [3] and the required processes are entirely OR-based. Using IMPLY would require many more steps for these processes. Additionally, the ORs are not reused, making the reliability of the FELIX-OR irrelevant.

B. IMPLY-based Full Adders

Of IMPLY-based adders there are the three general types serial, parallel, and hybrid (semi-serial, semi-hybrid), of which the latter endeavors to produce a better balance between computing time and memristor usage. Serial adders consist of memristors in the same row or column of a crossbar array and

TABLE I: Truth Table for IMPLY and FELIX-OR functions

a	b	$a \rightarrow b$	$a \vee b$
1	1	1	1
1	0	0	1
0	1	1	1
0	0	1	0

TABLE II: Comparison of Exact n-Bit IMPLY Adders

Adder	Time Steps	Memristors
Serial [15]	22n	2n+3
Parallel [12]	5n+16	4n+1
Semi-Serial [24]	10n+2	2n+6
Semi-Parallel [13]	17n	2n+3

can only perform one operation per cycle [2], [15], [20]. The best State-of-the-Art (SoA) IMPLY-based full adder requires $22n$ steps and $2n + 3$ memristors for an $n \quad bit$ calculation [15]. Note that other logic variants such as MAGIC [21] or SIXOR [19] feature faster exact full adders [19], [23], however, we have chosen IMPLY due to its higher reliability, higher practical relevance, and more wide-spread use, as mentioned in Section II-A. Parallel adders consist of multiple rows that are not connected, which enables the parallelization of some operations [12]. Via switches, each row can be connected to a shared c-memristor, which serves for computing and propagating the carry-out. Not all steps can be parallelized due to the dependency on the previous carry-out. Adders of this type require $5n + 16$ steps and $4n + 1$ memristors for an $n \quad bit$ addition. The hybrid adders are either semi-serial which consists of two parallel rows with inputs [14], [24] or semi-parallel with two parallel rows, with one input and a work memristor each [13]. Semi-serial requires $10n + 2$ steps and $2n + 6$ memristors, and semi-parallel $17n$ steps and $2n + 3$ memristors for an $n \quad bit$ addition. Table II lists the SoA adders of each type together with their computing time and memristor requirements.

We chose the SoA serial adder by Rohani et al. [15] for our proof of concept implementation (ApprOchs) of the APAF due to the simplicity of its topology.

C. Approximate Computing

Approximate computing primarily involves modifying logic by removing or altering gates or individual transistors and redesigning the corresponding truth table, accepting some degree of inaccuracy in the process. This approach can enhance performance metrics like energy efficiency, chip area utilization, and computation speed. However, the trade-off is a reduction in computational accuracy [25], [26]. Due to the inherent nature of approximate computing, it is most suitable for error-tolerant applications, such as machine learning, image processing, and signal processing [5]–[7]. In machine learning, accuracy is a commonly used quality metric, whereas in image processing, the Peak Signal-to-Noise Ratio (PSNR) measures noise levels, with a PSNR above 30dB generally being acceptable in the SoA [27], [28]. Additional metrics like the Structural Similarity Index Measure (SSIM) are also used to evaluate image quality, considering the importance of structure for human visual perception [29].

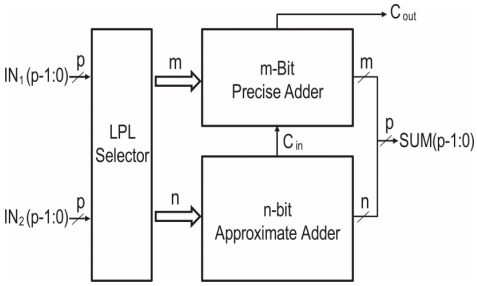
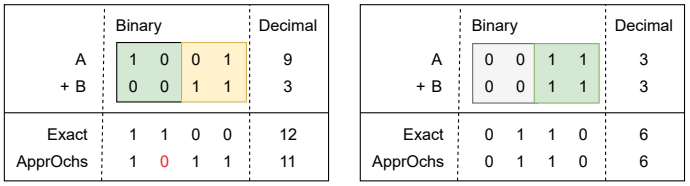


Fig. 1: CMOS Adaptive Approximate Adder [16].



(a) Case 1: Partial Approximation (b) Case 2: No Approximation

Fig. 2: Example additions using the APAF. Green sections indicate exact addition, yellow shows OR-approximated addition, and white represents performing no addition.

Various approximation techniques have been developed for CMOS technology [7], [16], [30]–[34] and others [35]. Recently, there has been growing interest in approximate full adders utilizing memristors, by employing the memristors into an approximate addition algorithm. This movement began with the introduction of Memristor Ratioed Logic-based approximations in [36] and [37]. More recent work has explored IMPLY-based approximate full adders in serial topology, as discussed in [10] and [38], which optimized the truth table using specific input vectors, leading to reductions in processing steps and energy consumption by up to 42% and 68%, respectively. In [39], a semi-serial topology was employed for an approximate adder, achieving reductions of up to 29% in steps and 34% in energy usage. By cleverly disregarding the carry, [40] achieved even better results for serial, parallel, semi-serial, and semi-parallel topologies. Other work [41] focuses on reducing latency instead of energy, achieving computation in only two steps per approximate bit. However, the aforementioned adders do not consider the nature of the data they are processing. These adders provide a static segment of bits to which the approximation is applied during runtime and cannot adapt based on the data. The only exception to this is the CMOS-based adaptive approximate adder shown in Figure 1 [16]. The adder first sequentially compares both inputs, starting at the Most-Significant Bit (MSB), for whether the inputs are equivalent or not. The XOR-based comparison stops at the first unequal bit and assigns all bits before that to the exact adder and all bits after that to the approximate adder. However, this method incurs a large overhead and is not directly translatable into memristive in-memory computing.

In this work, we propose the novel APAF, where the inputs are divided into segments whose sizes are determined by the user before runtime. During runtime the APAF enables the adders to discriminate based on the inputs which segment of bits should be approximated and which should be computed correctly or not at all for a superior balance between error, computational speed, energy consumption, and memristor requirements. The implementation of our APAF in the form of the ApprOchs adder offers significant improvements in speed and energy efficiency over the SoA, while maintaining or improving on error metrics for image processing and machine learning-based applications in most cases.

III. ApprOchs Design

The principle for our approximate adder, as is also true for numerous others, is that by reducing the amount of bits that are calculated accurately through changing the algorithm steps that are employed on the memristors to calculate the addition, we can reduce both processing time and energy consumption. Often, approximate adders have the bits that are approximated fixed, starting from the Least-Significant Bit (LSB) upwards. This has the advantage that the absolute error can be kept as small as possible, as only the lowest bits can be calculated inaccurately. For applications that rely on small integers, this poses an issue because the calculations may all be approximated, and the error increases with each further addition when the integer stays in the band of approximated bits, i.e. for small integers, the relative error is high. To solve this problem, our proposed adder uses an adaptive element that automatically determines the approximation depending on how large the number is. For this, a n-bit integer is split into two fixed segments before runtime, where the lower LSB part consists of k bit. It then checks the upper, MSB part of $n \quad k$ bit for whether any of the bits are equal to one by using an OR operation. If there is a large integer being added, the upper bit part of the addition will use an exact algorithm, while the lower bit part is added approximately by using OR operations. In this case, there are no carries propagated in the lower part (especially not between the sections), making both parts completely parallelizable.

For example, Figure 2a shows the 4-bit addition of two integers values, namely, 9 and 3. The example demonstrates that approximation is used in the lower part and accurate calculation is used in the upper part. When using an exact adder, the result is 12, as shown in Figure 2a. Using the ApprOchs adder, the 4-bit integers are split into an upper bit half (high bits) and a lower bit half (low bits), as shown in Figure 2a. The ApprOchs adder then checks if the upper half contains a 1 in any of its bits. As this is the case in this example (integer A's MSB is 1), the upper part will be added accurately, and the lower part will be "added" using OR operations. This produces 11 as the result. Checking the upper half for a 1 means that the ApprOchs adder does not work very well for two's complement (negative) numbers, because it will always select the upper part in those numbers as the MSB will always be 1. This is especially a problem as in 1, always all-bits equal to 1. Therefore, there is a large interpreted value discrepancy compared to unsigned integers.

If the condition that the upper bit part contains a bit equal to one is *not* met, the LSBs will be added using an exact algorithm, and the upper bit part will be ignored and not calculated at all, saving both power and time. This second case

TABLE III: Adaptive Precision Adder Framework Algorithm

Case 1 (Non-Zero MSBs): $1 \in a_{n:k+1}b_{n:k+1}$		
Time Step	$n-k$ **MSBs**	k **LSBs**
1	$OR = \bigvee_{i=k+1}^{n}(a_i \vee b_i)$	
2	$\mathrm{ex}(a_{n:k+1}; b_{n:k+1})_1$	$S_i = a_i \vee b_i \ \forall i \leq k$
3	$\mathrm{ex}(a_{n:k+1}; b_{n:k+1})_2$	
⋮	⋮	
$t_{ex}(n-k)+1$	$\mathrm{ex}(a_{n:k+1}; b_{n:k+1})_{t_{ex}(n-k)}$	

Case 2 (All-Zero MSBs): $1 \in a_{n:k+1}b_{n:k+1}$		
Time Step	$n-k$ **MSBs**	k **LSBs**
1	$OR = \bigvee_{i=k+1}^{n}(a_i \vee b_i)$	
2		$\mathrm{ex}(a_{k:1}; b_{k:1})_1$
3		$\mathrm{ex}(a_{k:1}; b_{k:1})_2$
⋮		⋮
$t_{ex}(k)+1$		$\mathrm{ex}(a_{k:1}; b_{k:1})_{t_{ex}(k)}$

is illustrated in Figure 2b. Here, 3 and 3 are added together, which when added exactly results in 6. This calculation can be seen in Figure 2b. As the upper half (high bit 1 and high bit 0) does not contain a 1, the ApprOchs adder does not calculate this part and only accurately calculates the lower half, saving half the energy and half the compute time. This case-distinguishing behavior has the benefit that for tasks relying on small integers, such as machine learning tasks, where the weights can be very small, the calculation is done exactly, and for larger integers, the LSBs are approximated to get a small relative error and achieve both the energy and time benefits of approximate computing. Another advantage of the ApprOchs adder's adaptive element is that it can be applied to different integer sizes. This makes it applicable to many use cases and different integer sizes. This system could also be double-ended or have more adaptive sections for larger integers. By default $k = \frac{n}{2}$, so there is an upper half and a lower half, but k can be set to any other value to suit a certain use case.

Figure 3 shows the circuit of the adder. For 8-bit, it uses 24 memristors in total: 16 memristors for the two 8-bit input integers, 2 working memristors for the exact addition, 1 memristor to store the result of the initial OR-operation to decide the case, k memristors for the k LSB OR-approximations (by default $k = 4$), and 1 carry-out memristor. The exact adder algorithm we use requires two working memristors, that is why our circuit features two working memristors as well.
The control circuit that decides the case (Case 1 or Case 2) is built using CMOS logic and switches depending on the state of the OR memristor. In the case of a logical 1 in the OR memristor, the control logic will switch the accurate algorithm to the high bits and enable the approximate algorithm for the low bits. For a logical 0 in the OR memristor, the control circuit will switch the accurate algorithm to the low bits and will disable the approximate algorithm.

Table III shows the general algorithm for the APAF using the exact adder labelled as ex. Any addition starts out with reading all values of the upper part of memristors into a single FELIX-OR (labelled OR) in step 1 shown in Table III. This OR-memristor acts as a control signal to turn the approximation either on or off. So, depending on this memristor, either the upper part will be calculated accurately and the lower part will be approximated, or the lower part will be added accurately, and the upper part will not be calculated at all. In the case where the initial FELIX-OR operation and consequently the memristor produces a logical 1, the circuit switches the upper part of the memristors to exact adder and the lower part memristors to the FELIX-OR blocks (labelled s) that are used to approximately add those LSB memristors. In Table III this is step 2 of case 1. This switching behavior is accomplished by using normally closed switches for the lower bit part so if $OR = 1$, it will cut off the accurate algorithm from the lower part and instead connect the OR blocks to the outputs of the memristors in step 2 as previously explained. The results of the approximate addition are therefore stored in the s-memristors, and the outputs of any exact addition, whether for the upper part or the lower part, are stored in the corresponding a-memristors. The carry-out of any exact addition is stored in the c-memristor. In step 2 of the algorithm, the exact adder commences its operation either on the MSBs or LSBs depending on the case and then runs for $t_{ex}(k)$ steps for a total of $t_{ex}(k)+1$ steps for the whole algorithm. The approximate computation of the LSBs in case 1 happens totally simultaneously in one step, in step 2.

IV. Experiments and Results

A. Hardware and Accuracy Criteria

For error analysis and accuracy assessment, we use the commonly used metrics Mean Error Distance (MED) [42], Normalized Mean Error Distance (NMED) [43], and Mean Relative Error Distance (MRED) [26] presented in Equations (1) to (3), respectively. In these equations, $Z_{Ex;i}$ and $Z_{Ax;i}$ represent the i-th of 2^{2n} possible n-bit sums computed with a specific exact and approximate adder, respectively. We adjusted Equation (3) from the literature slightly to skip the division by zero that would be caused by $z_{Ex;1}$, which is the sum of $0+0$ and the only number that equals 0 for the exact adder in the denominator.

$$MED = \frac{1}{2^{2n}} \sum_{i=1}^{2^{2n}} |Z_{Ex;i} - Z_{Ax;i}| \tag{1}$$

$$NMED = \frac{MED}{2^{n+1} - 1} \tag{2}$$

$$MRED = \frac{1}{2^{2n} - 1} \sum_{i=2}^{2^{2n}} \frac{|Z_{Ex;i} - Z_{Ax;i}|}{Z_{Ex;i}} \tag{3}$$

For efficiency comparisons, we evaluate speed as the number of steps required to perform a full n-bit addition, energy for an n-bit addition, and area requirements as measured by the number of memristors used.

B. Experimental Setup

To evaluate the performance in terms of energy consumption and to validate the correctness of the proposed algorithms, we conducted circuit simulations using LT-SPICE. These simulations were based on the Voltage-controlled ThrEshold Adaptive Memristor (VTEAM) model [44], which was implemented in SPICE [14]. This approach enhances the reliability of our simulations and ensures their practical relevance. Additionally,

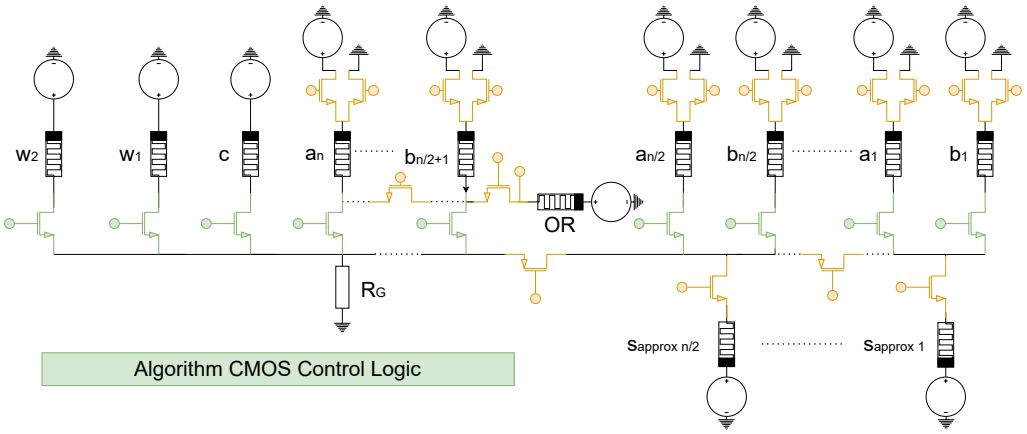


Fig. 3: Circuit of an n-bit ApprOchs adder with approximation level k. NMOS and PMOS represent normally opened and normally closed switches, respectively. All yellow circles represent lines directly connected to control and apply the OR logic operation, whereas green circles represent connections to the algorithm control logic. To reduce clutter in the figure connecting lines are not shown.

employing this widely used model facilitates a more straightforward comparison with previous studies, such as those in SoA literature [15], [38], [39]. To evaluate the accuracy metrics for different configurations of n and k, we use a model in Python on all possible 2^{2n} input combinations, which for $n = 8$ is 65536.

C. Results

Speed: The time ApprOchs requires to compute an n-bit addition largely depends on the time of the underlying exact adder t_{ex}. Since the potential approximation of the LSB part can be performed in parallel to the exact adder ex and only requires one step, it does not contribute to the overall time of the algorithm. However, before the exact adder may begin operation, performing the FELIX-OR over the MSBs requires an additional step. Therefore, in the general case, the step count t amounts to

$$t(n) \quad \frac{\max(k; n \quad k)}{n} \quad t_{ex}(n) + 1; \tag{4}$$

where $t_{ex}(n)$ is the time required by the underlying exact adder, n is the number of bits in the addition and k is the approximation split, i.e., the number of LSBs comprising the lower part. The upper part then consists of $n \quad k$ bit. Equation (5) shows the equivalent formulation. Furthermore, when using the serial adder [15] as an exact adder, the step count is given by Equation (6).

$$t(n) \quad t_{ex} \quad \max(k; n \quad k) + 1 \tag{5}$$

$$t(n) \quad 22 \quad \max(k; n \quad k) + 1 \tag{6}$$

For $k \leq \frac{n}{2}$, the step count t varies depending on the specific input. To obtain predictable operation times, the control logic will need to wait for the slowest case. Therefore, for the general case and in case of using this adder in a crossbar scenario, the inequality becomes an equality:

$$t(n) = t_{ex} \quad \max(k; n \quad k) + 1: \tag{7}$$

Table IV shows these slowest cases for $n = 8$ and $k < n$. For the default case of $k = \frac{n}{2}$, the equation simplifies to Equation (8). When using the serial adder [15] as an exact adder, it further simplifies to Equation (9). For speed, we only report relative performance numbers because real latency is technology dependent, which limits how well we can compare our work to others using a different technology. In the literature, it is common to report speed values in terms of the number of cycles as opposed to absolute latency values in seconds. Furthermore, this will ensure that our work remains comparable in the future as we move to newer technology.

$$t(n) = \frac{1}{2} t_{ex}(n) + 1; \tag{8}$$

$$t(n) = 11n + 1: \tag{9}$$

Area: Equation (10) depicts the number of memristors required, where n and k are as in Equation (4) and $sp_{w;ex}(n)$ is the number of additional work memristors of the underlying exact adder (usually 2 or 3 memristors). The $2n$ originate from the two n-bit inputs, k for the LSB approximation, one as carry-out, and one as initial FELIX-OR (labelled OR in the circuit). The number of memristors required for the exact serial adder is $sp_{ex}(n) = 2n + 3$ and with $n = 8$ it is $sp_{ex}(8) = 19$. Table IV shows the area for an 8-bit ApprOchs adder with a serial exact adder and various k.

$$sp(n) = sp_{w;ex}(n) + 2n + k + 2; \tag{10}$$

Energy: The energy consumed by the ApprOchs adder with the underlying serial exact adder [15] was measured using LT-SPICE. The energy consumption in nJ is

$$e(n; k)_{a;b} = 0{:}202(n \quad k) + \begin{cases} e_1 & \text{if } 1\ 2\ a_{n:k+1} b_{n:k+1}, \\ e_2 & \text{otherwise,} \end{cases}$$
$$\text{where } e_1 = 4{:}0789(n \quad k) + 0{:}210k; \quad e_2 = 4{:}0789k; \tag{11}$$

where a, b are arbitrary n-bit inputs and e_1, e_2 correspond to the energies in nJ in cases 1 and 2, respectively. In Case 1, there

TABLE IV: Step Count, Area, Energy, and Accuracy for 8-bit ApprOchs using [15] as Exact Adder ($t_{ex}(8) = 176$, $sp_{ex}(8) = 19$, $e_{ex}(8) = 32.631$). For Energy and Accuracy, the mean over all possible inputs (mixed cases) and over each case are reported. Accuracy for Case 2 is omitted since it does not produce any errors.

Mode	Time Steps		Mem-ristors	Energy (nJ)			Accuracy					
							Mixed			Case 1		
	$t(k)$	$t_x(k)$		Mixed	Case 1	Case 2	MED	NMED	MRED	MED	NMED	MRED
Exact	$t_{ex}(8) + 1$	177	**20**	34.246	-	-	0.0	0.0	0.0	-	-	-
k=1	$\frac{7}{8}t_{ex}(8) + 1$	155	21	30.174	30.176	5.493	0.2511	0.00049	0.00135	0.251	0.00049	0.00136
k=2	$\frac{6}{8}t_{ex}(8) + 1$	133	22	26.101	26.105	9.370	0.7472	0.00146	0.00396	0.750	0.00153	0.00374
k=3	$\frac{5}{8}t_{ex}(8) + 1$	111	23	22.025	22.035	13.247	1.7542	0.00344	0.00906	1.848	0.00351	0.00993
k=4	$\frac{4}{8}t_{ex}(8) + 1$	**89**	24	17.960	17.964	17.124	3.7240	0.00731	0.01818	3.757	0.00732	0.01917
k=5	$\frac{5}{8}t_{ex}(8) + 1$	111	25	14.003	13.893	21.001	7.6487	0.01502	0.03447	7.759	0.01513	0.0292
k=6	$\frac{6}{8}t_{ex}(8) + 1$	133	26	**10.762**	9.822	24.877	14.650	0.02878	0.05722	15.750	0.03076	0.04897
k=7	$\frac{7}{8}t_{ex}(8) + 1$	155	27	11.501	5.751	28.754	23.662	0.04648	0.07312	31.755	0.06201	0.07547

are $n - k$ exact adder steps and k FELIX-OR approximations, while in Case 2, there are k exact adder steps. Case 1 is triggered when there is at least a single one in either of the input number's $n - k$ MSBs. Additionally, irrespective of the case, a FELIX-OR is performed over the $n - k$ MSBs to initially determine the case.

Other work has simply averaged the energy over all possible input combinations. To make this work comparable, we will do the same. However, since our algorithm works differently depending on the inputs, we need to employ a weighted average of the input combinations for Case 1 (upper bits exactly, lower bits approximately) and Case 2 (upper bits not calculated, lower bits exactly). The mean energy over uniformly distributed inputs in nJ is thus

$$e(n;k)_{a;b} = 0.202(n - k) + \frac{(2^{2n} - 2^{2k})e_1 + 2^{2k}e_2}{2^{2n}} \quad (12)$$

where e_1 and e_2 are as in Equation (11) and inputs a and b are from a uniform distribution over the possible input combinations. The weighted average (the fraction in Equation (12)) is calculated based on how many combinations trigger Case 1 and how many trigger Case 2, assuming uniform input distribution to ease comparison. Please note that for $k < n - 2$, the following becomes true: $2^{2k} << 2^{2n}$. Therefore, uniformity-specialized Equation (12) can be approximated for smaller ks as

$$e(n;k)_{a;b} - 4.2809(n - k) + 0.210k \quad (13)$$

We report energy in absolute values since it is heavily technology-dependent and there is no common method to report relative energy unit for energy consumption (unlike for clock cycles for speed and number of memristors for area). Table IV shows the energy for $n = 8$ and various k using [15] for the exactly computed bits, where standalone [15] requires $e_{ex}(8) = 4.0789n = 32.631nJ$. Additionally, it shows the energy assuming inputs are either exclusively from Case 1 or Case 2.

Accuracy: To evaluate the accuracy, we calculated MED, NMED, and MRED for an 8-bit adder by applying all 2^{16} possible input combinations. Furthermore, we calculated the error metrics assuming only input combinations leading to Case 1, with Case 2 being practically 0 in all metrics due to no errors (exact calculations). The results are displayed in Table IV. We observe all three metrics rising together with k. This is because k increases the highest number that is approximated. Another reason is that the higher k is, the more numbers are approximated, as can be derived from Equation (12).

V. Comparison and Discussion

In this section, we compare our proposed approach to the SoA algorithms found in the literature, primarily focusing on the approximated adders from [38], [45], and [39], with additional comparisons drawn from the most recent works [40]. The comparative results, covering energy consumption, speed, area, and accuracy, are summarized in Tables V and VI. All comparisons are performed with $n = 8$ and $k = 5$ unless otherwise noted. We also added how ApprOchs performs if inputs are taken from either case exclusively (ApprOchs Case 1, ApprOchs Case 2).

A. Energy Consumption

Our proposed approach, ApprOchs, exhibits significantly lower energy consumption compared to the SoA. Specifically, ApprOchs requires 14.003 nJ, which is 39.2% less than SIAFA 1,3 and SIAFA 4 from [38], 32.9% less than SAID 1 [41] and 38.4% less energy than the SAFAN adder from [45]. In comparison to the proposal of [39], ApprOchs requires 32.4% less energy. Furthermore, the energy consumption of ApprOchs is also 26% lower than SINC and SINC+ [40], and 10% less than the semi-parallel S-SINC implementation [40] which previously required the lowest energy. Despite also using OR-based approximation in the lower k bit, ApprOchs has the edge over S-SINC [40] because ApprOchs can skip the computation of the upper $n - k$ bit for some possible input combinations (2^{2k} of 2^{2n} possible combinations skip the upper bits). These results demonstrate that ApprOchs is highly energy-efficient.

B. Speed

The speed of ApprOchs is comparable to the SoA. Our method takes 111 steps, which is on par with the SIAFA adders from [38] and SAFAN [45]. However, it is outperformed by the SINC and SINC+ implementations [40], which take 48 steps, but at much higher energy consumption. ApprOchs is generally faster than the serial exact adder [15] it is made of,

TABLE V: Energy, Speed, and Area Comparison

Algorithm	Energy (nJ)		Speed		Area	
	e(n,k)	e(8,5)	t(n,k)	t(8,5)	s(n,k)	s(8,5)
Serial Exact [15]	$4.0789n$	32.6311	$22n$	176	$2n + 3$	**19**
Parallel Exact [12]*	$4.0772n$	32.6176	$5n + 18$	58	$4n + 1$	33
Semi-Serial Exact [24]*	$3.8435n + 0.8053$	31.5580	$10n + 2$	82	$2n + 6$	22
Semi-Parallel Exact [13]*	$4.8339n$	38.6712	$17n$	136	$2n + 3$	**19**
SIAFA 1,3 [38]*	$1.7090k + 4.8250(n - k)$	23.0200	$8k + 22(n - k)$	106	$2n + 3$	**19**
SIAFA 2 [38]*	$2.5131k + 4.8250(n - k)$	27.0405	$10k + 22(n - k)$	106	$2n + 3$	**19**
SIAFA 4 [38]*	$1.7066k + 4.8250(n - k)$	23.0008	$8k + 22(n - k)$	106	$2n + 3$	**19**
SAFAN [45]*	$1.6628k + 4.8250(n - k)$	22.7890	$7k + 22(n - k)$	101	$2n + 3$	**19**
[39]*	$1.66k + 3.84(n - k) + 0.86$	20.7345	$5k + 10(n - k) + 3$	58	$2n + 6$	22
SINC [40]*	$0.7230k + 4.8250(n - k)$	18.9900	$3k + 22(n - k) + 3$	84	$3k + 4(n - k) + 1$	28
SINC+ [40]*	$0.72k + 4.82(n - k) + 0.78$	18.8744	$3k + 22(n - k) + 3$	84	$3k + 4(n - k) + 2$	29
PINC [40]*	$0.723k + 4.0772(n - k)$	15.8466	$5(n - k) + 18$	**33**	$3k + 4(n - k) + 1$	28
PINC+ [40]*	$0.72k + 4.07(n - k) + 0.78$	18.9900	$5(n - k) + 18$	**33**	$3k + 4(n - k) + 2$	29
S-SINC [40]*	$0.57k + 3.84(n - k) + 1.06$	15.4566	$2k + 10(n - k) + 3$	45	$2n + 6$	22
S-SINC+ [40]*	$0.57k + 3.84(n - k) + 1.87$	16.2590	$2k + 10(n - k) + 5$	45	$2n + 6$	22
S-PINC [40]*	$0.6372k + 4.8339(n - k)$	17.6877	$3k + 17(n - k) + 3$	66	$2n + 3$	**19**
S-PINC+ [40]*	$0.63k + 4.83(n - k) + 0.92$	18.6164	$3k + 17(n - k) + 2$	68	$2n + 3$	**19**
SAID 1 [41]	$1.2283k + 4.8250(n - k)$	20.6165	$2k + 22(n - k)$	76	$2(n - k) + 2k + 3$	**19**
SAID 2 [41]	$1.5488k + 4.8250(n - k)$	22.2190	$6k + 22(n - k)$	96	$2n + k + 3$	24
ApprOchs	Equation (12)	**14.003**	$22 \times \max(k; n - k) + 1$	111	$2n + k + 4$	25
ApprOchs C. 1**	Equation (11)	13.8837	$22 \times (n - k) + 1$	67	$2n + k + 4$	25
ApprOchs C. 2**	Equation (11)	21.0005	$22 \times k + 1$	111	$2n + k + 4$	25

(*) Values taken from [40], which uses the same experimental setup.
(**) Assuming input distributions only resulting in that case.

TABLE VI: Accuracy Comparison

Algorithm	n=8, k=1			n=8, k=2			n=8, k=3			n=8, k=5		
	MED	NMED	MRED	MED	NMED	MRED	MED	NMED	MRED	MED	NMED	MRED
SIAFA1, 3 [38]	0.25	0.0004	0.0013	0.875	0.0017	0.0048	2.062	0.004	0.0115	8.8554	0.0173	0.0522
SIAFA2 [38]	0.25	0.0004	0.0013	1	0.0019	0.0055	2.656	0.0052	0.015	13.498	0.0264	0.0822
SIAFA4 [38]	0.5	0.0009	0.0027	1.25	0.0024	0.0068	2.625	0.0051	0.0145	10.6562	0.0208	0.0616
Proposal of [39]	0.5	0.0010	0.0027	1.1250	0.0022	0.0062	2.2500	0.0044	0.0125	8.9121	0.0174	0.0514
No Carry [40]	0.25	0.00049	0.0013	0.75	0.0015	0.0040	1.75	0.0034	0.0092	7.75	0.0152	0.0377
No Carry+ [40]	0.25	0.00049	0.0013	**0.625**	**0.0012**	**0.0034**	**1.375**	**0.0027**	**0.0073**	**5.875**	**0.0115**	**0.0293**
SAID 1 [41]	0.5000	0.0020	0.0028	1.2500	0.0049	0.0069	2.6250	0.0103	0.0147	10.6562	0.0418	0.0614
SAID 2 [41]	0.5000	0.0020	0.0028	1.1250	0.0044	0.0063	2.1875	0.0086	0.0124	8.5293	0.0334	0.0519
ApprOchs	0.25	0.00049	0.0013	0.747	0.0014	0.0039	1.754	0.0034	0.009	7.6487	0.01502	0.0344
ApprOchs Case 1*	0.25	0.00049	0.0013	0.75	0.0015	0.0040	1.75	0.0034	0.0092	7.75	0.0152	0.0377
ApprOchs Case 2*	0	0	0	0	0	0	0	0	0	0	0	0

(*) Assuming input distributions only resulting in that case.

and also than the semi-parallel exact adder [13]. While other approximate algorithms, like S-SINC and S-SINC+ [40], exhibit faster speeds (taking only 45 steps), these algorithms generally exhibit higher energy consumption. Therefore, ApprOchs offers a balanced trade-off between speed and energy efficiency, particularly for applications where energy consumption is the priority. It should also be noted that with a faster underlying adder, like the parallel exact adder [12], better speeds are to be expected. Furthermore, the fastest configuration for ApprOchs with the serial exact adder currently is $k = 4$ at 89 steps. However, please note that this fastest configuration features worse energy characteristics than for instance $k = 6$, if the input data is uniformly distributed.

C. Area Usage

In terms of area, ApprOchs demonstrates moderate resource usage with $2n + k + 4$ memristors required for implementation. This is notably better than the parallel exact adder [12] and the SINC, SINC+, PINC, and PINC+ [40] variants, but also recognizably worse than the best the SoA can offer with, e.g., the SIAFA [38], SAFAN [45], SAID 1 [41], and serial exact adders [15]. Overall, ApprOchs is rather efficient in terms of area and very efficient in terms of energy, making it a suitable candidate for low-power applications.

D. Error Metrics

We also compared ApprOchs to the SoA using error metrics such as MED, NMED, and MRED, as shown in Table VI. ApprOchs performs exceptionally well, achieving similar accuracy to SIAFA [38] for small values of k, but significantly outperforming these approaches as the approximation degree increases. For $n = 8$ and $k = 5$, ApprOchs achieves an NMED of 0.01502, which is 14% lower than SIAFA 1,3 [38] and the approach presented in [39], achieving the second-best value. Moreover, the MRED for ApprOchs is 0.0344, which is just outperformed by No Carry+ [40]. No Carry+ [40] is able to outperform ApprOchs because it estimates a carry-in for the $n \quad k$ highest bit, which gives it an edge over ApprOchs. On the other hand, No Carry [40] performs ever so slightly worse compared to ApprOchs, despite essentially using the

same approximation mechanism of OR-ing the lower k bit, because ApprOchs computes exactly for 2^{2k} of 2^{2n} possible input combinations. This advantage increases with higher k, since more numbers are affected. It is still not enough to offset the carry-in from No Carry+ in terms of errors, but it makes enough of a difference in terms of energy to beat the SoA. These results highlight the robustness of ApprOchs in minimizing approximation errors, particularly for higher degrees of approximation.

Despite outperforming the SoA in terms of energy requirements and achieving excellent error characteristics, arguably the main benefit of ApprOchs, namely its ability to adjust to the input data, has not been sufficiently explored, yet. The following sections will examine the behavior of the ApprOchs adder in real-world tasks with non-uniform input distributions. To this end, we test ApprOchs in two machine learning and two image processing use cases. To enable a comprehensive comparison, we implement an 8-bit signed multiplier based on binary long multiplication, as described in [46]. The 8-bit multiplier produces a 16-bit product. However, we implemented it using an 8-bit adder for the partial product paired with shifting and or'ing instead of a 16-bit adder. The underlying adder is either the ApprOchs adder or the exact adder from [15].

VI. Application in Machine Learning

To evaluate our proposal in the context of machine learning, we chose an application in deep learning using the handwritten digits dataset MNIST with an 8-bit signed integer quantization of AlexNet and an application in classical machine learning using k-means clustering on an artificial dataset.

A. Quantized AlexNet on MNIST

Even though AlexNet [47] is an older representative of the convolutional neural network family, it consists of several convolution layers paired with dense classification layers. These building blocks are still highly relevant in modern architectures as they make up the bulk of today's computer vision models and are also used in time series and natural language processing.

In Python, we simulate our approximate hardware (adder and adder-based long multiplier) by re-constructing the model architecture with custom convolution and dense layers using the approximate operators with the original quantized model weights. This way, we can test the degradation in model inference performance compared to the exact version. Furthermore, we examine different levels of approximation by approximating $k\ 2\ [1;6]$ bit, respectively. We then evaluate the results for inference, accuracy and energy consumption on 1000 random samples of the MNIST dataset. Table VII shows the results of ApprOchs compared to the SoA approximate adders PINC, PINC+, SSINC, and SSINC+ [40], and SAID 1, SAID 2 [41]. We chose these adders for comparison because they are among the designs with the best speed, energy, and error characteristics in the literature, as shown in Tables V and VI. Figure 4 additionally provides a graphical overview of the results of Table VII. Up until $k = 5$, all adders except SAID 1, SAID 2 [41] can sustain the accuracy of the exact adder [15] of 98.9%. This demonstrates the error-tolerance of the network. The design goal for SAID [41] was speed over energy and accuracy and it shows in the benchmarks. SAID [41] performs worst in accuracy as well as energy consumption. For $k = 6$ and $k = 7$, the performance starts to degrade for the remaining adders. The exact computation [15] features the worst energy consumption behavior at 1726 mJ per image. The ApprOchs adder features by far the best energy behavior throughout all approximation levels. The best energy consumption is achieved by the ApprOchs adder with $k = 1$ at 338 mJ per image (19.6% of the exact adder; 21.6% of the best SoA adder SSINC [40]) and perfect accuracy, thus showing the best overall performance. As opposed to the SoA approximate adders from the literature, the energy consumption of ApprOchs rises together with k in this specific application, while the SoA approximate adders require less energy per level of k. Analysis confirmed that the low energy consumption of ApprOchs, especially for low k, arises from the Rectified Linear Unit (ReLU) activation function, which maps negative outputs to 0. Therefore, the next layer features a convolution with many inputs as zero, which prompts the adaptive adder ApprOchs to only require the LSBs to perform the costly long multiplication and therefore merely uses a fraction of the energy of other adders. We computed that 90.94% of the activations are zero in the exact network, while only 9.05% of activations are greater than zero. Furthermore, we observed that the weights in AlexNet [47], as is common with neural networks, are normally distributed around zero (see Figure 5), which leads to many weights being either 0 or 1 (or other small integers) for which the calculation of the LSBs suffice. The reason why the adder with $k = 1$ produces such a high accuracy is because two's complement negative numbers are handled more accurately. For a normal distribution of 8-bit numbers, the mean relative multiplicative error is 0.26 for $k = 1$ while it is 3.61 for $k = 4$. This is due to the negative numbers, especially $-1 = (11111111)_2$, which suffer extremely when only the upper 4 bits are calculated exactly as opposed to the upper 7. ApprOchs' energy distribution shown in Figure 6 further illustrates how important the input data is to ApprOchs' success in this use case: The standard deviation decreases with rising ks because the energy discrepancy closes between the favorable case (Case 2) and the unfavorable case (Case 1). If the input data is favorable and Case 2 is taken, a lot more energy is conserved with lower ks than with higher ks. Additionally, if the unfavorable Case 1 is taken, the cost is much higher for lower ks than for higher ks. Therefore, with higher ks this gap diminishes and the standard deviation falls. This use case illustrates excellently how adaptive behavior can be utilized for superior performance in use cases where data is favorably distributed.

B. k-means Clustering

As a second use case, we choose k-means clustering as it is a simple, effective and relevant algorithm for clustering and unsupervised machine learning. Similarly to AlexNet (see Section VI-A), we simulate the approximate hardware in Python and implement the k-means algorithm with the k-means++ method for centroid initialization. As the distance metric, we use Manhattan distance over Euclidean distance to

TABLE VII: Digit prediction accuracy (Acc.) in percentage and consumed energy (E) per image in mJ for different approximation k and the latest SoA approximate adders. Note that PINC (PINC+) and SSINC (SSINC+) have the same error behavior. For ApprOchs, we have additionally provided standard deviation of energy (SD E).

Approx.	ApprOchs			PINC [40]		PINC+ [40]		SSINC [40]	SSINC+ [40]	SAID 1 [41]		SAID 2 [41]	
	Acc.	E	SD E	Acc.	E	Acc.	E	E	E	Acc.	E	Acc.	E
Exact [15]	**98.9**	1726	-	-	-	-	-	-	-	-	-	-	-
k=1	**98.9**	**338**	39.5	**98.9**	1568	**98.9**	1657	1567	1589	**98.9**	1946	**98.9**	1954
k=2	**98.9**	**419**	33.4	**98.9**	1481	**98.9**	1568	1481	1502	**98.9**	1851	97.9	1868
k=3	**98.9**	**499**	28.9	**98.9**	1395	**98.9**	1479	1394	1416	98.7	1756	82.2	1781
k=4	**98.9**	**578**	22.3	**98.9**	1308	**98.9**	1391	1308	1329	97.2	1661	38.3	1695
k=5	**98.9**	**657**	16.7	**98.9**	1222	**98.9**	1302	1221	1243	76.8	1565	13.1	1608
k=6	98.8	**737**	9.7	98.7	1135	**98.9**	1213	1135	1156	31.7	1470	12.6	1522
k=7	**97.9**	**820**	**4.7**	97.6	1049	**97.9**	1124	1048	1070	12.9	1375	10.2	1436

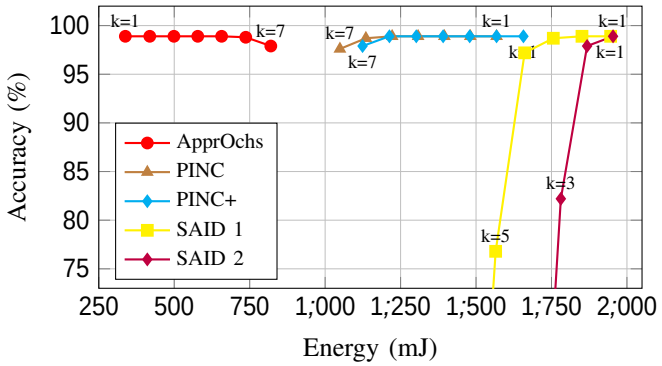


Fig. 4: Energy-Accuracy plot of selected SoA adders.

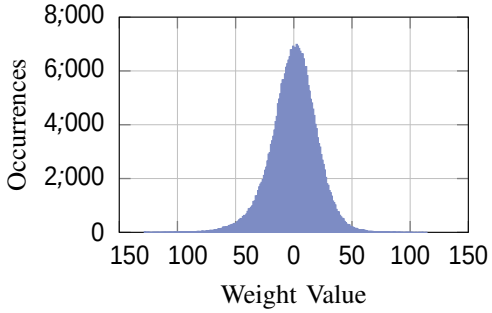


Fig. 5: Weight Distribution of 8-bit Quantized AlexNet.

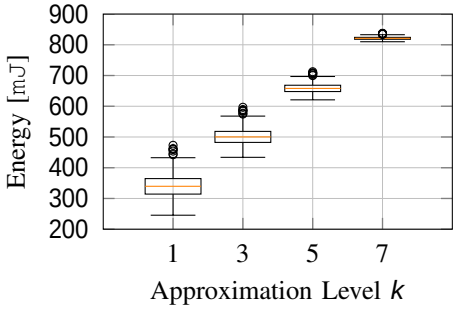


Fig. 6: Statistical distribution of energy consumption for for ApprOchs, $k \in \{1;3;5;7\}$. The circles indicate outliers.

TABLE VIII: Clustering accuracy and its standard deviation in percentage, average total consumed energy and its standard deviation in mJ, and average iterations for exact and approximate strategies.

Method	Acc. (%)	SD Acc.	ø E (mJ)	SD E	ø Iter.
Exact [15]	88.82	**7.64**	2.161	0.360	**5.72**
k=1	90.50	7.98	1.928	0.513	7.25
k=2	90.89	7.87	1.687	0.388	8.64
k=3	**91.83**	8.03	1.431	0.320	7.02
k=4	90.94	8.14	1.158	0.228	8.19
k=5	87.69	9.32	0.848	0.267	12.78
k=6	75.77	11.65	**0.670**	**0.212**	18.94

remove the need for a custom square root implementation using approximate adders. To be able to compare the accuracy values of the exact and approximate versions, we generate artificial test data with ground-truth labels. The data samples consist mainly of Gaussian blobs with a varying number of cluster centers and diverse variances. The data also includes some anisotropically distributed data and ring structures. The data is quantized to 8-bit signed integers. The dataset consists of 36 samples with 400 vectors each. We then calculate the accuracy of the exact hardware and with $k \in [1;6]$ bit approximation with respect to the ground-truth labels. Since the initial centroid selection is random, we cycle the permutations of the predicted labels in order to find the highest accuracy for each result. Additionally, we specify the number of expected clusters correctly, as was used in the ground-truth. This is done because we do not want to evaluate the clustering algorithm itself, but rather the difference between an exact and approximate implementation. We report accuracy with respect to ground truth and energy consumption for configuration. For each configuration, we specify the maximum number of iterations for the k-means clustering algorithm as 30. If the clusters do not converge by then, the calculation is stopped and results are evaluated as-is.

Table VIII shows the results of the adders tested. Accuracy-wise, the exact and $k \in [1;4]$ versions perform similarly. The configuration with $k = 6$ performed significantly worse. The best accuracy is achieved by $k = 3$, which yields a higher accuracy even than the exact adder. The best energy (but worst accuracy) is achieved by the $k = 6$ configuration. It requires only 31% of the energy of the exact adder. For $k \in [1;4]$, the average iterations required for convergence are about 50% higher than for the exact version and for $k = 6$, they are 230% higher. With so many more iterations and thus computations, it is surprising that $k = 6$ still presents results that far outperform the SoA in terms of energy consumption. The many more computations must thus be compensated for by the high approximation level.

Figure 7 depicts several samples from the test data with the ground truth and the result of each tested configuration. The pictures additionally show the classification accuracy for that

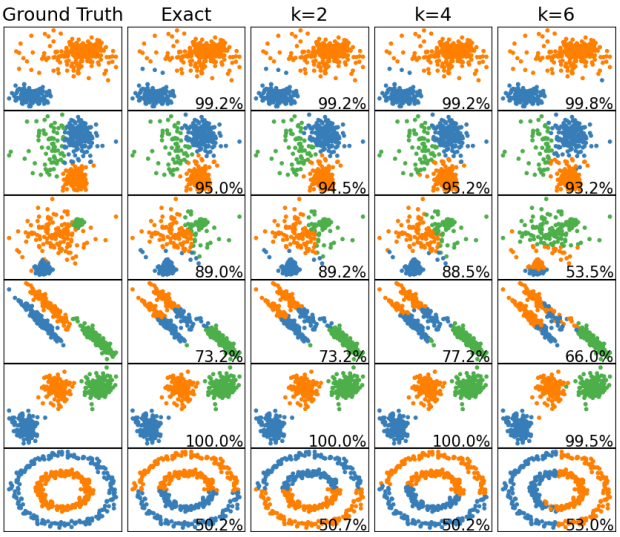


Fig. 7: Examples of clustering with different k. The left-most column shows the ground-truth labels. Each example contains the accuracy with respect to ground truth in percent.

sample with respect to the ground truth. Especially samples 3, 4 and 6 demonstrate how the generated data breaks the assumptions made by the k-means algorithm. None of the adders perform particularly well in these examples, but the relative difference between exact and approximated versions is negligible. Approximation levels $k = 5$ and $k = 6$ show generally worse performance. This is most likely due to the convergence strategy of k-means. It iterates as long as the cluster centroids keep changing.

With $k = 6$, negative numbers will always be approximated in the LSBs due to the sign bit. This will often cause the centroids to keep changing slightly due to the approximation in the LSBs. Therefore, the algorithm does not converge quickly and is eventually stopped by the maximum iterations, ending prematurely. This explains the high average iteration count. The energy is relatively low because only 2 bits are calculated exactly compared to the exact adder, where 8 bits are calculated exactly.

The absolute accuracy values should be considered more leniently compared to the AlexNet case study, because the ground truth itself leaves room for interpretation and is not as well-defined as with the human-annotated MNIST dataset. The better comparison of error behavior is probably given by the average number of iterations until convergence, which rise with each further approximation level. However, the accuracy values and that approximate adders can outperform exact ones to a certain degree show that approximate computing is very well suited for k-means clustering due to its high error-tolerance and fuzzy correct solution which is inherent to unsupervised learning.

In summary, the results of both machine learning use cases show how the distribution of the data can be exploited to achieve lower energy consumption than with non-adaptive adders. The AlexNet use case has an especially favorable data distribution for the ApprOchs adder, while the k-means was evaluated on a largely uniform data distribution. ApprOchs was designed for high-accuracy low-energy additions adapted to the input data. The machine learning use cases show that the ApprOchs meets these design goals and is able to outperform SoA adders by a large margin in terms of energy use.

VII. Application in Image Processing

We evaluated the performance of the ApprOchs in image processing tasks, specifically in image blurring and edge detection. Both operations are based on image convolution, but differ in their kernels and effects: the blurring kernel (K_1 in Figure 8) contains positive elements and produces a smoothing effect that reduces the impact of high-frequency noise, whereas the edge detection kernel (K_2 in Figure 8) incorporates negative elements and identifies edges, which is useful for feature extraction, segmentation and object recognition. Image convolution involves the element-wise multiplication of the kernel with image pixels, summing the products, and normalizing by the kernel's weight sum. To manage sums exceeding the 8-bit range, a 16-bit ApprOchs adder was utilized, with multiplications performed using binary long multiplication. Energy consumption for the 16-bit design was estimated based on the 8-bit implementation.

$$K_1 = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \qquad K_2 = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Fig. 8: Blurring (Gauss) kernel (K_1) and edge detection (y-Sobel) kernel (K_2).

A. Evaluation Data Set

Image blurring and edge detection were performed at different approximation levels k on a dataset of 100 images, each sized 256×192. The dataset was generated by sampling images from the training set of Google's OpenImagesV7 database [48], followed by resizing and conversion to grayscale.

B. Quality Metrics

We compare images produced using an exact adder ($k = 0$) to those produced by approximate adders ($k > 0$). Variations in image quality are attributed to the applied approximation.

Image quality is measured using Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM). PSNR measures image quality by comparing the noise level or pixel intensity between an original and modified version of an image. SSIM, on the other hand, evaluates the perceived structural similarity and integrity between those two images.

C. Image Blurring

Figure 9 presents an example of image blurring using an exact adder ($k = 0$) and the ApprOchs adder with progressively increasing approximation level k. At the initial approximation levels, the output is virtually indistinguishable from the exactly-computed reference image. However, at $k = 4$, a noticeable degradation in quality begins, with the loss in image quality

TABLE IX: Median image quality (PSNR in dB, SSIM) and energy consumption (mJ) for 16-bit image blurring with ApprOchs adder at varying approximation levels k.

Method	PSNR [dB] ↑	SSIM ↑	Energy [mJ] ↓
Exact	∞	1.000	240.36
k=1	58.28	0.999	216.92
k=2	50.82	0.997	187.28
k=3	43.74	0.995	161.69
k=4	36.81	0.985	142.04
k=5	30.84	0.952	128.43
k=6	25.12	0.880	121.82
k=7	20.27	0.788	123.80
k=8	19.05	0.759	134.30

becoming more pronounced as the approximation level k increases. Despite the decline in quality, the scene remains intact, even under the most extreme approximation.

The measurements in Table IX confirm these observations, presenting median image quality in PSNR and SSIM and energy consumption for various approximation levels on the evaluation image set. Median image quality declines steadily with increasing approximation, remaining high up to $k = 3$.

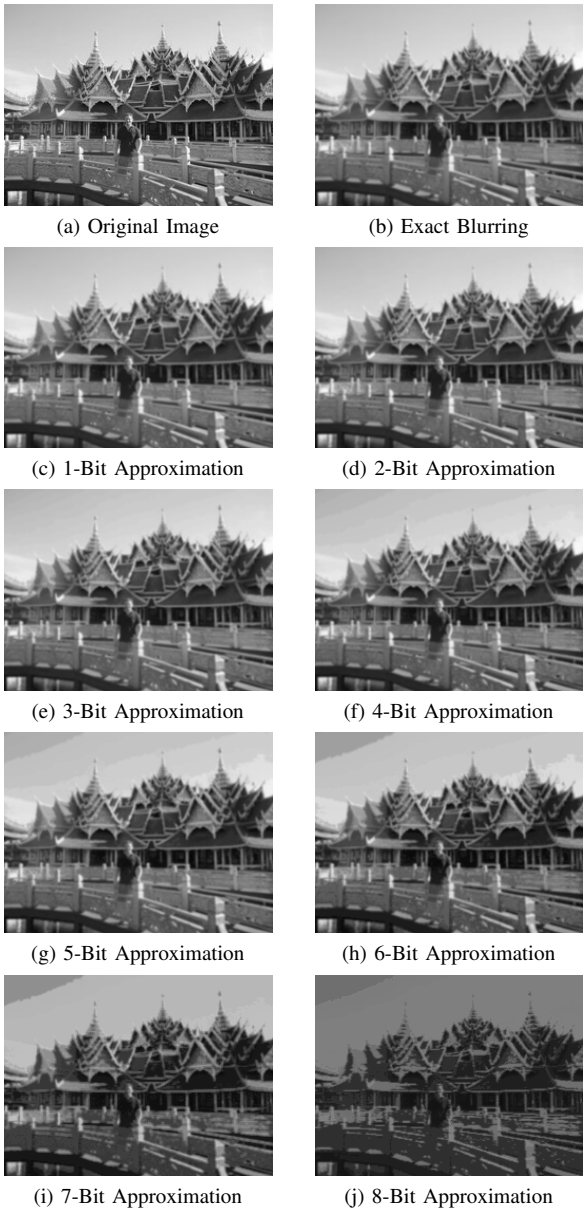
Beyond this, quality degradation becomes noticeable, with significant drops occurring at $k > 5$, where SSIM falls below 0.9 and PSNR below 30 dB. Median energy consumption decreases as k increases, reaching a minimum at $k = 6$, after which it rises slightly. This is likely because, at higher k, there are fewer operand combinations where the MSBs can be omitted from computation, leading to a slight increase in energy consumption.

Figure 10 provides a comparison of multiple SoA adder designs, including ApprOchs, showing how energy consumption and PSNR image quality scale with approximation level k. For clarity, less energy-efficient designs with overlapping image quality data curves were excluded, retaining only the most competitive designs in terms of energy efficiency.

The energy consumption plot reveals that ApprOchs has the lowest energy consumption compared to all other designs across all approximation levels and it is the only examined design with a non-linear energy consumption trend - a result of its decision mechanism. Additionally, as k increases, fewer computations can be omitted, causing energy consumption to stabilize and the deviation to approach zero.

Furthermore, the PSNR plot shows a stable, near-linear trend for ApprOchs, starting at a very high image quality of about 58dB and maintaining strong signal preservation as k increases. Although S-SINC+ holds a 10dB advantage for $3 \quad k \quad 7$, ApprOchs consistently outperforms most competing designs in terms of PSNR image quality.

Figure 11 relates the mean PSNR to the corresponding mean energy consumption. ApprOchs forms a pareto front between $k = 1$ and $k = 6$, demonstrating its energy efficiency. Notably, S-SINC+ (with $k = 1$) is the only other design to break this front in the high quality region, achieving 62.2dB. ApprOchs consistently delivers the same PSNR quality with noticeably lower energy consumption compared to other designs, underscoring its superior energy efficiency.



(a) Original Image (b) Exact Blurring (c) 1-Bit Approximation (d) 2-Bit Approximation (e) 3-Bit Approximation (f) 4-Bit Approximation (g) 5-Bit Approximation (h) 6-Bit Approximation (i) 7-Bit Approximation (j) 8-Bit Approximation

Fig. 9: Image quality degradation in image blurring with increasing approximation level k: original image, blur using an exact adder ($k = 0$), and blurs with progressively higher k based on a 16-bit ApprOchs adder.

The most promising ApprOchs configuration for image blurring is $k = 3$, providing excellent image quality at around 43dB (PSNR) and 0.995 (SSIM) with significant energy savings. Compared to the exact adder ($k = 0$), energy consumption is reduced by 32.7%. Relative to the first practical reference ($k = 1$), PSNR decreases by 24.9%, and SSIM by only 0.004, making $k = 3$ a favorable trade-off. Additionally, ApprOchs at $k = 3$ uses 13.4% less energy than the second-most efficient design, S-SINC+ (at $k = 5$), while maintaining comparable image quality at roughly 43dB (PSNR).

D. Edge Detection

Figure 12 presents an example of edge detection using an exact adder ($k = 0$) and the ApprOchs adder with progressively

TABLE X: Median image quality (PSNR in dB, SSIM) and energy consumption (mJ) for 16-bit edge detection with ApprOchs adder at varying approximation levels k.

Method	PSNR [dB] ↑	SSIM ↑	Energy [mJ] ↓
Exact	∞	1.000	173.90
k=1	12.13	0.833	164.00
k=2	6.65	0.459	153.28
k=3	5.19	0.117	144.80
k=4	5.41	-0.14	138.27
k=5	5.90	0.042	134.07
k=6	5.54	0.005	131.84
k=7	5.08	0.005	133.05
k=8	3.48	0.007	136.31

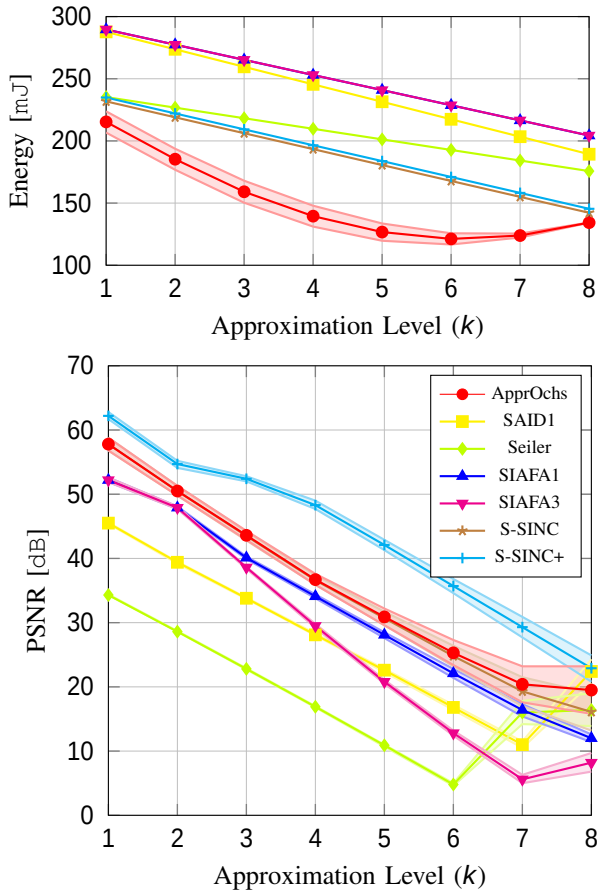


Fig. 10: Mean energy consumption and PSNR of ApprOchs and SoA designs plotted against the approximation level k. The shades represent the standard deviation of the curve. Certain designs were excluded for the sake of clarity, if they yielded the same image quality but were less energy-efficient.

increasing approximation level k. At $k = 1$, the output closely matches the exact reference, though visual artifacts are noticeable. For $k > 1$, visual quality declines rapidly; structures remain discernible up to $k = 4$, but beyond this, noise overwhelms the image.

The measurements in Table X confirm these findings, showing median image quality (PSNR, SSIM) and energy consumption for various approximation levels. The results reveal that the ApprOchs design is unsuitable for handling

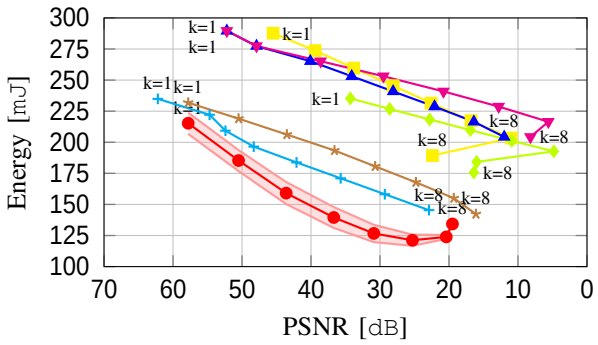


Fig. 11: Mean energy consumption of ApprOchs and SoA designs plotted against mean PSNR. The error regions represent one standard deviation from the energy mean. First and last approximation level k have been marked because of non-linear development of energy consumption.

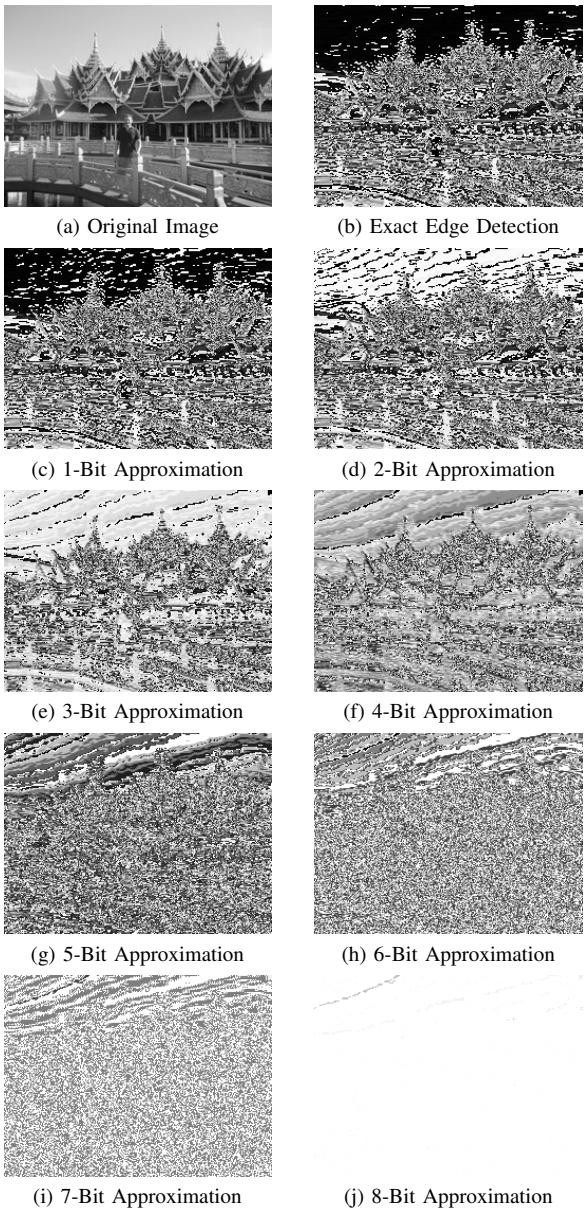
negative operands, as image quality is unacceptable across all approximation levels. This issue stems from the design's reliance on the sign bit in its decision mechanism, leading to undesirable computational behavior, rapid quality degradation, and negligible energy savings. Therefore, we recommend against using ApprOchs for operations involving negative operands and did not conduct further analysis or comparisons for edge detection.

VIII. Conclusion

In this work, we presented *ApprOchs*, a memristor-based adaptive approximate adder, suitable for in-memory computing. The design incorporates a decision mechanism that analyzes the bit patterns of the operands to determine which parts of the addition should be computed exactly, approximately, or skipped entirely, thereby reducing power consumption and improving speed. We evaluate the adder for two machine learning applications (MNIST and k-means) and for two image processing applications (image blurring and edge detection).

The ApprOchs adder demonstrates an improvement in energy characteristics, with a 9.5% efficiency gain over the currently best SoA in-memory approximate adder in the general case of uniformly distributed data. It achieves this while scoring second-best in terms of error behavior, with only a 22% higher MED at $k = 3$ compared to the best SoA adder. Due to its design, ApprOchs also shows advancements in terms of time complexity (see Equation (4)), making it nearly twice as fast as its exact counterpart at an approximation level of $k = 4$, all while remaining competitive in terms of hardware cost. Yet, its flexibility and efficiency makes the ApprOchs adder design well-suited for a wide range of error-tolerant applications, such as machine learning and image processing on low-energy embedded devices, IoT devices such as wearables, or on remote sensors.

When comparing ApprOchs to both exact and approximate adders from the literature, we observe significant improvements in several metrics. For instance, when applied to the MNIST classification task, ApprOchs achieves a 78.4% reduction in energy consumption compared to the best SoA approximate



(a) Original Image (b) Exact Edge Detection

(c) 1-Bit Approximation (d) 2-Bit Approximation

(e) 3-Bit Approximation (f) 4-Bit Approximation

(g) 5-Bit Approximation (h) 6-Bit Approximation

(i) 7-Bit Approximation (j) 8-Bit Approximation

Fig. 12: Image quality degradation in edge detection with increasing approximation level k: original image, edge detection using an exact adder ($k = 0$), and edge detections with progressively higher k based on a 16-bit ApprOchs adder.

adders, while maintaining the same accuracy as the exact adder at 98.9%. Similarly, for k-means clustering, ApprOchs provides a 69% energy saving with no degradation in clustering quality over the exact adder. In the image blurring application, ApprOchs reduces energy consumption by 32.7% compared to its exact counterpart. Its most promising configuration ($k = 3$) achieves an excellent image quality, with 43.74dB PSNR and 0.995 SSIM, while consuming 13.4% less energy than the most energy-efficient competing SoA design (S-SINC+) offering comparable image quality. However, in the edge detection application, the performance is unfavorable, even with $k = 1$, as indicated by a median PSNR rating of 12.13dB and median SSIM score of 0.833. This behavior probably stems from ApprOchs' difficulty in handling negative numbers, where sign bits interfere with the decision mechanism.

In future work, we plan to enhance the ApprOchs adder design by expanding the decision mechanism to support larger data types, optimizing the exact addition process, and addressing the current limitations in handling negative signed numbers. More specifically, we plan on employing exact adders based on other logic variants such as MAGIC [21] or SIXOR [19] or other SoA exact adders such as [19], [23], to improve ApprOchs' speed. These future improvements will help to scale accuracy, reduce energy consumption and extend the range of applications that can benefit from ApprOchs. Furthermore, we plan to conduct an analysis of our design in a crossbar setup to evaluate its throughput.

REFERENCES

[1] L. Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.

[2] E. Lehtonen and M. Laiho. Stateful implication logic with memristors. In *2009 IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 33–36. IEEE, 2009.

[3] S. Gupta *et al.* Felix: Fast and energy-efficient logic in memory. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7. IEEE, 2018.

[4] H. Kim *et al.* Memristor-based multilevel memory. In *2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010)*, pp. 1–6. IEEE, 2010.

[5] N. TaheriNejad and S. Shakibhamedan. Energy-aware adaptive approximate computing for deep learning applications. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 328–328, 2022.

[6] S. Bharti *et al.* Review of approximate computing in image processing applications. In *2022 4th International Conference on Artificial Intelligence and Speech Technology (AIST)*, pp. 1–6. IEEE, 2022.

[7] V. Gupta *et al.* Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, 2013.

[8] V. K. Chippa *et al.* Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–9, 2013.

[9] C. Li *et al.* In-memory computing with memristor arrays. In *2018 IEEE International Memory Workshop (IMW)*, pp. 1–4, 2018.

[10] S. E. Fatemieh *et al.* Approximate in-memory computing using memristive imply logic and its application to image processing. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 3115–3119, 2022.

[11] V. Gupta *et al.* Impact: Imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pp. 409–414, 2011.

[12] A. Karimi and A. Rezai. Novel design for a memristor-based full adder using a new imply logic approach. *Journal of Computational Electronics*, 17(3):1303–1314, Sep 2018.

[13] S. Ganjeheizadeh Rohani *et al.* A semiparallel full-adder in imply logic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):297–301, 2020.

[14] N. TaheriNejad *et al.* A semi-serial topology for compact and fast imply-based memristive full adders. In *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, 2019.

[15] S. G. Rohani and N. TaheriNejad. An improved algorithm for IMPLY logic based memristive full-adder. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–4, April 2017.

[16] S. Kim and Y. Kim. Adaptive approximate adder (a 3) to reduce error distance for image processor. In *2016 International SoC Design Conference (ISOCC)*, pp. 295–296. IEEE, 2016.

[17] D. B. Strukov *et al.* The missing memristor found. *Nature*, 453:80–83, May 2008.

[18] J. Borghetti *et al.* 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464:873–876, April 2010.

[19] N. TaheriNejad. Sixor: Single-cycle in-memristor xor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(5):925–935, 2021.

[20] S. Kvatinsky *et al.* Memristor-based material implication (imply) logic: Design principles and methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2054–2066, 2014.