

Article

Designing Approximate Reduced Complexity Wallace Multipliers

Ioannis Rizos ^{*,†}, Georgios Papatheodorou [†] and Aristides Efthymiou 

Department of Computer Science & Engineering, University of Ioannina, 45110 Ioannina, Greece; g.papatheodorou@uoi.gr (G.P.); efthym@uoi.gr (A.E.)

* Correspondence: ioa.rizos@uoi.gr

[†] These authors contributed equally to this work.

Abstract: In the nano-scale era, enhancing speed while minimizing power consumption and area is a key objective in integrated circuits. This demand has motivated the development of approximate computing, particularly useful in error-tolerant applications such as multimedia, machine learning, signal processing, and scientific computing. In this research, we present a novel method to create approximate integer multiplier circuits. This work is based on a modification of the well-known Wallace tree multiplier, called the Reduced Complexity Wallace Multiplier (RCWM). Approximation is introduced by replacing conventional Full Adders with approximate ones during the partial product reduction phase. This research investigates the characteristics of 8×8 -, 16×16 -, and 32×32 -bit Approximate Reduced Complexity Wallace Multipliers (ARCWM), evaluating their accuracy, area usage, delay, and power consumption. Given the vast search space created by different combinations and placements of these approximate Adders, a Genetic Algorithm was used to efficiently explore this space and optimize the ARCWMs. The resulting ARCWMs have an area reduction of up to 65% and a power consumption reduction of up to 70%, with no worse delay than the RCWM. Multipliers created with this method can be used in any application that requires parallel multiplication, such as neural accelerators, trading accuracy for area and power reduction. Additionally, an ARCWM can be used alongside a slow shift-and-accumulate multiplier trading off accuracy for faster calculation. This methodology provides valuable guidance for designers in selecting the optimal configuration of approximate Full Adders, tailored to the specific requirements of their applications. Alongside the methodology, we provide all of the tools used to achieve our results as open-source code, including the Register-Transfer Level (RTL) code of the 8×8 -, 16×16 -, and 32×32 -bit Wallace Multipliers.

Keywords: approximate computing; arithmetic circuits; Wallace Multiplier; non-dominated multi-objective genetic algorithms; design space exploration



Academic Editor: Costas Psychalinos

Received: 23 December 2024

Revised: 9 January 2025

Accepted: 13 January 2025

Published: 16 January 2025

Citation: Rizos, I.; Papatheodorou, G.; Efthymiou, A. Designing Approximate Reduced Complexity Wallace Multipliers. *Electronics* **2025**, *14*, 333. <https://doi.org/10.3390/electronics14020333>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

A critical report issued by the Semiconductor Industry Association and the Semiconductor Research Corporation warns that, without significant advancements in energy-efficient computing, the energy demands of computing systems could surpass the world's projected energy production by 2040 [1]. This demand has driven the development of various energy-aware computing techniques. Among these, the Approximate Computing (AC) paradigm has garnered significant attention over the past two decades. AC proves especially beneficial for error-tolerant applications, including multimedia, machine learning, signal processing, and scientific computing.

In this research, the well-known Wallace tree multiplier [2], which was modified by Waters into the Reduced Complexity Wallace Multiplier (RCWM) [3], is utilized as the foundation for developing an approximate parallel multiplier. The approximation is incorporated during the partial product reduction phase by substituting conventional Full Adders with approximate ones, resulting in the development of the Approximate Reduced Complexity Wallace Multiplier (ARCWM).

The RCWM was selected for this work over other multipliers for two key reasons. First, compared to other designs, including the well-known and widely used Wallace tree multiplier, relatively little research has been conducted on approximating the RCWM. Second, the RCWM primarily relies on Full Adders, and uses very few Half Adders. With a wide variety of approximate Full Adder designs available in the literature, the RCWM seamlessly leverages these resources. The RCWM can be employed in all applications where a Wallace tree multiplier is traditionally used, including digital signal processing, machine learning, embedded systems, and other fields requiring high-speed multiplication. Moreover, it can directly replace the Wallace tree multiplier in any application, since they differ only in the partial product reduction stage.

This work investigates the characteristics of 8×8 -, 16×16 -, and 32×32 -bit ARCWMs, assessing their accuracy, area utilization, delay, and power consumption. The examination of a range of multiplication sizes using our proposed methodology allows for a well-rounded evaluation of the merits of the methodology. Given the vast search space created by different combinations and placements of these approximate Adders, a Genetic Algorithm (GA) was employed to efficiently explore this space and optimize the ARCWMs. A GA is inspired by the process of natural selection, based on which it solves optimization problems [4,5]. The Non-Dominated Multi-objective Genetic Algorithm-II (NSGA-II) [6] is an elitist multi-objective GA, which uses a fast non-dominated sorting approach. Being multi-objective, this algorithm is able to conduct optimization on multiple objective functions at the same time and, as such, it is more fitting for this problem because it allows for accuracy, area, delay and power of the ARCWMs to be optimized simultaneously.

In our previous work, we studied 8×8 ARCWM multipliers for Field-Programmable Gate Arrays (FPGA) [7] and 32×32 multipliers for Application-Specific Integrated Circuits (ASIC) [8]. In this work, we complete the study by adding 8×8 and 16×16 multipliers for ASICs. This allows us to perform a comparison regarding the scaling and effectiveness of the method for multiplying numbers of different sizes. This novel methodology provides valuable guidance for designers in selecting the optimal configuration of an ARCWM tailored to the specific requirements of their applications. Alongside the methodology, we provide all the tools used to achieve our results as open-source code, including the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) code of the 8×8 -, 16×16 -, and 32×32 -bit Wallace Multipliers [9].

This methodology can be used to generate ARCWMs to be used in error-tolerant applications where multiplication accuracy can be traded off for savings in area, delay, and power consumption. Such applications could be cases where parallel multiplication is desired, such as neural accelerators. The ARCWM approach allows the designer to implement multiple fast multipliers into their circuit without sacrificing a large amount of area. Furthermore, an ARCWM could be utilized in a system with tight area budget, alongside a slow shift-and-accumulate exact multiplier. The exact multiplier is smaller in size, and can be used for calculations which are not error tolerant, while the ARCWM can be used for the calculations which are error tolerant, speeding them up significantly.

The rest of the article is organized as follows. In Section 2, we discuss the reasons that have led to the Approximate Computing paradigm, and then we briefly review the techniques used in the paradigm. In Section 3, we discuss the Wallace tree multiplier,

the variation made by Waters [3], and we present the Approximate Reduced Complexity Wallace Multiplier. The evaluation method and the optimization are presented in Sections 4 and 5, respectively. The results of the evaluation of the Approximate Reduced Complexity Wallace Multiplier are presented and discussed in Section 6. Related work is presented in Section 7. Section 8 discusses topics important to the domain of evolving approximate circuits using GAs. Finally, Section 9 concludes this article.

2. Approximate Computing

The rapid advancements in semiconductor technology, which were guided for decades by Dennard's [10] and Moore's laws [11], are now facing significant limitations. Dennard's scaling law, which described the proportional improvement in performance and energy efficiency with transistor miniaturization, has ceased to hold true for several years, leaving a gap in technological progress. At the same time, Moore's law, which predicts the doubling of transistors per unit area every two years, is approaching its physical limits [12], if they have not already been surpassed.

These challenges underscore the need for novel approaches to the design and operation of computing systems. Approximate Computing emerges as a promising alternative, aiming to balance performance, energy consumption, and accuracy, offering solutions that address modern demands.

Approximate Computing intentionally modifies accuracy, defined as the deviation between approximate and exact results, to design systems that are faster, energy-efficient, and more compact. It capitalizes on the tolerance for minor inaccuracies in applications like image and video processing, artificial intelligence, and data mining, as well as the inherent imprecision of electronic sensors and the limits of human perception. At this point, it is also important to clarify the concept of error as it is used in Approximate Computing. According to [13], the term error is used to indicate that the output result is different from the accurate result produced with conventional computing. Accuracy differs from precision [13], which refers to the ability to distinguish between closely spaced discrete values. Precision is not related to errors in Approximate Computing, but rather to quantization noise introduced during the conversion of real values to digital representations.

In the past two decades, various terms have been used in the literature to describe the Approximate Computing paradigm. Briefly, we could mention the terms Relaxed Programming [14], Best-Effort Computing [15], Scalable Effort Design [16] and Approximate Computing [17]. It should be noted that, according to [18], Approximate Computing and Probabilistic/Stochastic Computing are distinct.

Regarding the term Approximate Computing, a number of definitions have been formulated. According to [17], Approximate Computing exploits the gap between the accuracy required by the applications/users and that provided by the computing system to achieve diverse optimizations. In [19], the author claims that Approximate Computing is based on

“the idea that we are hindering the efficiency of the computer systems by demanding too much accuracy from them”.

More recently, an alternative definition of the term has been proposed in [13]:

“Approximate Computing constitutes a radical paradigm shift in the development of systems, circuits & programs, build on top of the error-resilient nature of various application domains, and based on disciplined methods to intentionally insert errors that will provide valuable resource gains in exchange for tunable accuracy loss.”

In this article, we also propose the following definition, emphasizing a more human-centric perspective: Approximate Computing is a computing paradigm that mirrors the human brain's ability to perform imprecise calculations while maintaining functional accuracy. In terms of efficiency, Approximate Computing leverages inexact hardware and software to achieve significant gains in power, timing, and area, much like the brain's operation. Neurons, circuits, and neural codes in the brain are inherently optimized to conserve space, materials, time, and energy [20]. Furthermore, the brain operates through an internal "approximate number system" (ANS) [21], allowing it to process and make decisions based on the representation and manipulation of non-symbolic numerical quantities. Similarly, just as Approximate Computing relies on inexact data, primarily from sensors, the brain processes information from the human sensory system, which has limited capabilities, to perform computations and make decisions.

Approximate Computing Techniques

Over the past two decades, numerous approximate techniques have been introduced in the literature and classified in various ways. In this work, we will adopt the categorization proposed in [13], in which the approximate computing techniques are divided into two primary groups: software approximation techniques and hardware approximation techniques, each with several subcategories.

Software approximation techniques are divided into six sub-categories summarized as follows: Selective Task Skipping, Approximate Memorization, Relaxed Synchronization, Precision Scaling, Data Sampling, and Approximate Programming Languages. Software approximation techniques share common features such as approximation libraries and frameworks, compiler extensions, accuracy tuning tools, runtime systems, and language annotations.

Hardware approximation techniques operate at the lower levels of the computational stack, aiming to optimize circuit area, power consumption, and performance. These circuits form the building blocks of accelerators, processors, and computing platforms. In the literature, several libraries for approximate arithmetic circuits are available, including ApproxAdderLib [22], SMApplixLib [23], and EvoApprox8b [24], the last of which is used in [8] and in the present study.

The techniques are typically divided into three main categories: (1) Over-Clocking (OC), (2) Voltage Over-Scaling (VOS), and (3) Circuit Functional Approximation (CFA). In approximate hardware, two types of errors are identified: timing errors, which arise from VOS and OC, and functional errors, which result from CFA.

This study employs the CFA approximation technique, which is briefly outlined here for context. CFA is commonly applied to arithmetic circuits. The technique simplifies the original circuit design by employing a number of different methods: modification of the circuit's truth table, use of an approximate version of the original hardware algorithm, integration of inexact components as building blocks, or the application of approximate circuit synthesis.

3. Wallace Tree Multiplier

The Wallace tree multiplier [2] was introduced in 1964 by Australian computer scientist Chris Wallace. It is an efficient hardware design for binary multiplication. The process is executed in three main steps: partial product generation, partial product grouping and reduction, and final addition. The concept of Wallace tree multiplication is illustrated in Figure 1. It generates partial products by multiplying each bit of the multiplier with each bit of the multiplicand using the AND operation, resulting in N^2 partial products for two N -bit integer numbers. In order to simplify the computation, it uses Half Adders and Full

Adders in stages, to systematically reduce the operands (partial products and/or Adder's sum and carry) to just two rows.

At each stage, the Wallace Multiplier groups the operands into sets of three. The three rows are summed using Full Adders, while Half Adders are utilized when a column contains only two operands. The sum and carry signals generated by the half and Full Adders are forwarded to the next stage. If a single operand exists in a column, then it passes to the next stage too. This process is repeated iteratively until all partial products are summed. In the final stage, the resulting sum and carry from the last stage are added together using a high-speed carry-propagate Adder.

3.1. Reduced Complexity Wallace Multiplier

The Reduced Complexity Wallace Multiplier is an enhanced version of the original Wallace Multiplier, introduced by Waters and Swartzlander in 2010 [3]. Since Half Adders do not decrease the number of operands functioning as two-to-two compressors, RCWM reduces the use of Half Adders compared to the original multiplier. Instead, it slightly increases the use of Full Adders to achieve more effective compression of operands. In the modified design, the partial products are arranged in an inverted pyramid structure, as shown in Figure 1B. However, unlike the Standard Wallace method, columns with only two bits are not processed, but are instead carried forward to the next stage. Single bits are also passed to the next stage, as in the standard Wallace reduction tree. Half Adders are used selectively, only when necessary to ensure that the number of stages do not exceed those of the original Wallace multiplier. Specifically, Waters and Swartzlander proposed using Half Adders in a stage only if, in the previous stage, the number of rows (R) satisfies the condition $R \bmod 3 = 0$.

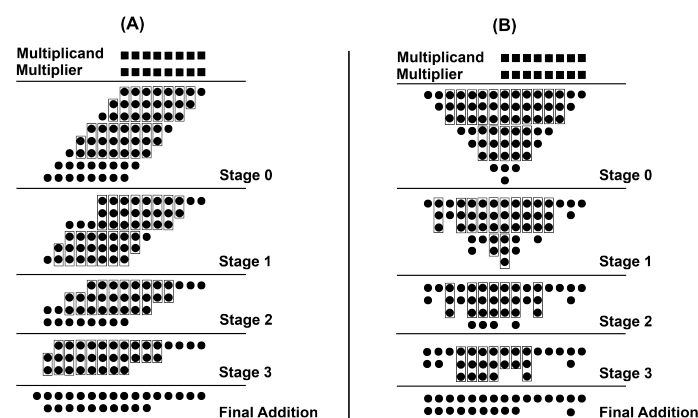


Figure 1. (A) 8×8 Wallace tree multiplier, (B) 8×8 Reduced Complexity Multiplier.

To facilitate the writing of HDL code during the development of [7] and [8], we created the RCWM in tabular form for 8-bit, 16-bit, and 32-bit numbers. The table includes the multiplier, the multiplicand, and the partial products, as well as all the Full Adders and Half Adders used. Following the algorithm, it illustrates the division into stages and the grouping of operands into sets. To simplify its use, each stage, group, and column is assigned a unique identifier which, in combination, serves as a code to indicate the Adder operating at each point, along with the signals it processes and generates. These tables act as wiring diagrams accompanying the VHDL code provided as part of this work [9].

3.2. Approximate Reduced-Complexity Wallace Multiplier

Building upon the RCWM, the Approximate Reduced Complexity Wallace Multiplier introduces further modifications to the reduction tree. To do this, every Full Adder (FA)

within the partial products reduction tree of the RCWM is treated as a slot with three inputs and two outputs. Within a slot, any 3-to-2 combinational circuit can be placed to stand in the place of a FA. In the ARCWM the slots initially occupied by conventional FAs in the RCWM are replaced by Approximate Full Adders (AFAs). The Half Adders (HAs) of the RCWM remain intact. To clarify this modification, an abstract representation of an 8×8-bit ARCWM is depicted in Figure 2. For the positions of the Full Adders, slots in the reduction tree are represented as squares numbered from 00 to 38, while the Half Adders are depicted as triangles.

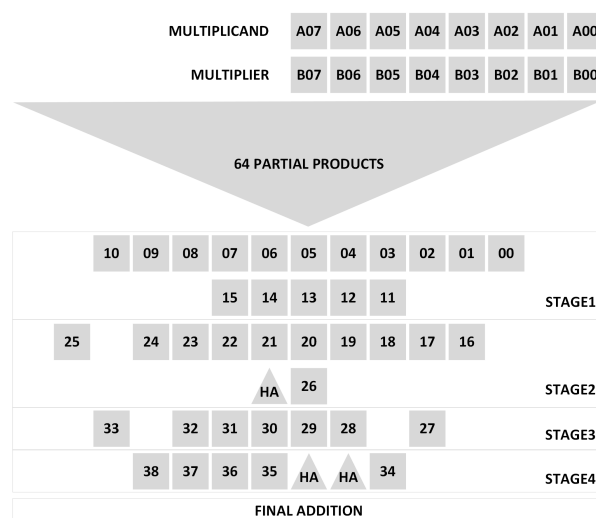


Figure 2. Abstract representation of an 8×8-bit ARCWM.

The state-of-the-art AFAs are developed by approximating the Sum and Carry outputs of the exact Full Adder, which are defined as follows:

$$Sum = A \oplus B \oplus Cin \quad (1)$$

$$Cout = A \cdot B + B \cdot Cin + A \cdot Cin \quad (2)$$

These approximations are applied to either the Sum, the Carry, or both outputs of the exact Full Adder. We extend the AFAs introduced in [25] by incorporating three new designs, resulting in a total of 28 distinct AFAs utilized in this study. Consistent with the naming convention from [25], the acronym ‘AFA’ is the same as in this work, and is followed by a number to index different versions. The new designs adopt this ‘AFA’ label, followed by Roman numerals (I, II, III) to distinguish them.

Based on the output being approximated, the AFAs organized and presented in Tables 1–3, according to this classification: those where the approximation is applied only to the Sum (AFA2, 3, 4, 5, 6, 7, 16, 16, 24, 25, 26, and 27), those where it is applied only to the Carry (AFA13, I, II, and III), and those where it is applied to both the Sum and the Carry (AFA1, 8, 10, 11, 12, 15, 17, 19, 20, 21, 22, and 23). It should be noted that in this work, three AFAs described in [25]—AFAs 9, 18, and 28—are not utilized, as they share the same Sum and Carry with AFA6. Any of the 28 AFAs or the exact FA—meaning a total of 29 options—can be used to fill a slot in the ARCWM, introducing approximation to the multiplier. It is obvious that, in the case that an exact FA is utilized in every slot of the ARCWM, the resulting multiplier is identical to the RCWM.

Table 1. AFA's with approximated Sum and Carry.

AFA	Sum	Carry
AFA1	$\bar{A} + B \cdot C$	A
AFA8	$\bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot C$	$B + A \cdot C$
AFA10	$\bar{A} \cdot \bar{B} + \bar{B} \cdot \bar{C}$	
AFA11	$(\bar{A} + B) \cdot C$	A
AFA12	A	A
AFA15	$\bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C}$	$C + A \cdot B$
AFA17	$\bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C} + A \cdot B \cdot C$	$\bar{A} \cdot C + \bar{B} \cdot C + A \cdot B \cdot \bar{C}$
AFA19	$\bar{A} \cdot B + A \cdot \bar{B} + C$	$A \cdot B$
AFA20	$\bar{C} + A \cdot B$	$C + A \cdot B$
AFA21	$A \cdot \bar{C} + B \cdot \bar{C} + A \cdot B$	C
AFA22	B	A
AFA23	C	$A \cdot \bar{C} + B \cdot \bar{C} + A \cdot B$

Table 2. AFA's with approximated Carry.

AFA	Sum	Carry
AFA13	$(A \oplus B) \oplus C$	C
AFAI		$\overline{(A \oplus B) \oplus C}$
AFAII		A
AFAIII		B

Table 3. AFA's with approximated Sum.

AFA	Sum	Carry
AFA2	$\bar{A} \cdot (B + C) + B \cdot C$	$A \cdot B + B \cdot C + A \cdot C$
AFA3	$(\bar{A} + B) \cdot C$	
AFA4	$\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C} + A \cdot B \cdot C$	
AFA5	$\bar{A} + B \cdot C$	
AFA6	$\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C}$	
AFA7	$\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C$	
AFA14	$C + \bar{A} \cdot B + A \cdot \bar{B}$	
AFA16	$\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C}$	
AFA24	$\bar{A} \cdot \bar{B} + A \cdot B$	
AFA25	$\bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot C$	
AFA26	$A + B + C$	
AFA27	$A \cdot B \cdot C$	

4. ARCWM Evaluation

ARCWMs are digital circuits, thus they can be evaluated in terms of the physical size required for the implementation of the multiplier (area), the time required for processing input operands and generating the resulting output (delay), and the power consumption

of the circuit (power). Area allows the designer to judge whether an ARCWM can be utilized in their design and then choose the quantity of ARCWMs to use based on their circuit size constraints. Delay is useful when designing the different clock regions and pipelining the design. Finally, power gives an indication of the power budget that the multiplier will require when added to the final design. Additionally to the physical circuit characteristics, calculated through synthesis, accuracy is also a helpful characteristic. Since CFA is used, ARCWMs are expected to produce mathematical errors in their results and, thus, accuracy is a measure of the expected produced errors. This section describes how these characteristic values are calculated.

The characteristics of area, delay, and power are tied to the underlying process technology used during manufacturing. In order to measure these values, synthesis and implementation is conducted for each ARCWM utilizing Cadence Genus. Genus elaborates the VHDL code of the ARCWM, and then synthesizes, optimizes and implements a physical circuit. This is performed based on the 180 nm GF180MCU open-source process development kit provided by Google and GlobalFoundries [26]; specifically, we utilize the seven-track Standard Cells library under operating conditions of 25 °C and 1.8 V. The choice of this library was made for ease, and no other specific reason. Ideally, a designer using the ARCWM method, when optimizing, will use the standard cell library on which their circuit is to be manufactured.

Synthesis optimization is deterministic: for the same input circuit, the output physical characteristics will always be the same. But, depending on the choice of constraints, the circuit performance will vary widely. In our experiments, when leaving the circuit delay unconstrained (i.e., no delay constraint), Genus produced results with large delay and optimized power. When constraining the clock and setting synthesis effort to “low”, we obtained results that displayed high power usage and were optimized only up to the given clock (i.e., close to zero positive slack) with little deviation. When setting synthesis effort to “high”, the resulting circuits were better optimized for power, and showed a bit more deviation in clock slack. Finally, by constraining the clock to an unrealistically small value (1 ns), the results had high deviation in delay, negative slack that can be subtracted from the constraint to determine the actual delay, but high power usage. Area seemed to be only affected by synthesis effort, but to a smaller degree than delay and power. Thus, a designer using this methodology must ascertain and use the best constraints for their application. For this study, a delay constraint slightly smaller than the RCWM was set, and “high” synthesis effort was used, effectively sacrificing delay for better power usage. An argument for making this choice is that RCWM is already optimized for delay, being a tree multiplier and, as such, lower power consumption is more desirable. However, this might not be the case in a specific application.

To calculate the accuracy of an ARCWM, the designer must provide inputs to the circuit and observe the produced output. Thus, two sets can be obtained: a set of expected-correct values E , and a second set P containing produced values by the approximate circuit. These two sets have a 1 – 1 correspondence such that the i -th value E_i is the product of two numbers and P_i is the produced product of the same two numbers from an ARCWM. The Error Distance metric for that pair of numbers is the distance between them,

$$Errors = \{|E_i - P_i| \mid \forall i \in input_vectors\}. \quad (3)$$

where $input_vectors$ is a set of all input vector pairs. This way, the *Errors* set can be calculated and studied to assess accuracy. Usually, it is useful to reduce the *Errors* set to a single number, in order to signify accuracy intuitively. To this end, the simplest choice is

the mean of the set: Mean Error Distance (MED), which is also known as Mean Absolute Error (MAE),

$$MAE = \sum_{i=1}^N \frac{|E_i - P_i|}{N}. \quad (4)$$

The designer could also use any of the error metrics mentioned in [18], such as Hamming distance and Mean Squared Error, or develop their own method. In this study, since the general case is presented, MAE is selected. In a specific application, the designer should choose the error analysis method that best illustrates accuracy for that application.

The amount of values in the sets E and P is at maximum equal to the amount of all possible input pairs of an ARCWM. Since an ARCWM is a binary multiplier, the amount of all possible input pairs is dependent on the input bits of the circuit. For an $N \times N$ multiplier, the input bits are $2N$, which results in 2^{2N} possible combinations of input number pairs. Noting that an ARCWM does not abide by the commutative property of multiplication, swapping the order of input operands might produce a different output. As such, the sets E and P for an $N \times N$ ARCWM have, at most, 2^{2N} expected/produced values. This number increases exponentially with the amount of input bits, and is a big consideration during the validation of such circuits, because the exponential increase in input combinations circuits require longer times to fully validate.

In the case of accuracy, there is no need to fully validate the ARCWMs, because they will have errors in the output regardless. To shorten accuracy calculation times, a subset of the *Errors* set can be used. Furthermore, the use of a subset can help the designer guide the ARCWM to be more accurate on input cases that are more common for their application. Considering that an ARCWM will be used for a specific error tolerant application, it might be possible to analyze that application and isolate cases of input pairs that appear more commonly, in order to use those values to obtain the E and P sets. As we will show in Section 5, this choice can help the ARCWMs to become optimized for this application.

For ARCWMs, in order to minimize the accuracy calculation times, alongside using a subset of at most 10^7 uniformly distributed random input pairs, a functional simulation is used instead of a behavioral simulation. After validation of the behavior of the Exact RCWM, since all ARCWMs are based on that, it can be assumed that they are behaviorally valid. Consequently, the only aspect of ARCWMs that needs to be simulated is what output they produce for any given input pair. To achieve this, a compiled C++ program is used, where the designer passes as input the architecture of an ARCWM and a binary file containing all of the test number pairs and the program calculates the MAE for that ARCWM.

Using the setup described in this section, Cadence Genus for synthesis, and C++ for functional simulation, the characteristics of accuracy, area, delay, and power can be calculated. With multiprocessing, multiple ARCWMs can be evaluated by running their Genus and C++ processes in parallel. With this approach, three ARCWMs were evaluated in parallel, due to software license constraints. Table 4 shows the total time required for a full evaluation of an ARCWM for all input widths, achieved on a consumer-grade Intel Core i7-2600 four-core processor and 16 GB of RAM. The C++ code, alongside the VHDL code mentioned earlier in this section is available at [9].

Table 4. Average seconds per one full evaluation of 8×8 , 16×16 and 32×32 ARCWMs.

	8×8	16×16	32×32
seconds/evaluation	7.97	17.75	60.17

5. ARCWM Optimization

The ARCWM approach for approximate multipliers consists of multipliers with multiple AFA slots. Each slot can be filled in with one of 29 potential AFA options, as presented in Section 3.2 and in Tables 1–3. Thus, an ARCWM can be represented as an K -dimensional vector, where K is the quantity of AFA slots within the multiplier, and all elements in the vector are integers from 0 to 28. This results in a search space for potential ARCWM configurations that expands exponentially and is equal to 29^K . For 8×8 multipliers, the search space consists of 29^{39} possible ARCWMs; this amount is of order of magnitude 10^{57} . Even if one 8×8 ARCWM could be evaluated per nanosecond, a brute-force approach to find the best would require an amount of years in the order of magnitude 10^{40} . For 16×16 ARCWMs, the search space consists of 29^{201} possible Approximate Multipliers, and for 32×32 , it consists of 29^{907} . For any ARCWM with multiplication width bigger or equal to 8 bits, exhaustive evaluation of the search space is infeasible, even under the best assumption of one ARCWM evaluation per nanosecond, which is also not practically feasible, as seen in Table 4.

5.1. Genetic Algorithms

Due to the expansiveness of the search space, there is a need for a more systematic search approach that can blindly navigate the search space and converge on performant ARCWMs. GA [4,5] emerges as a compelling solution for handling discrete problems of this nature. A GA works in the following manner:

1. Generate an initial Population of random Individuals.
2. If the stopping condition has been reached, finish.
3. Select some Individuals from the Population to combine.
4. Combine some of the selected Individuals to produce Children, and add them to the Population.
5. Mutate some of the Individuals in the Population randomly.
6. Choose the top Individuals from the Population to keep, and remove the rest.
7. Repeat from step 2.

A GA works with a Population, which is an array of Individuals, each of which is a solution to the problem at hand. In the case of ARCWMs, an Individual is a configuration of the AFA slots of the corresponding ARCWM, which is one of the 29^N possible vectors mentioned in Section 4. Initially (step 1), the Population consists of random solutions, and the algorithm iteratively updates these solutions to make them better. Using random variables will guarantee that the initial Individuals are uniformly scattered across the search space. This uniform distribution of points is a requirement for good quality of solutions [4], since areas that are not accessed initially might end up being inaccessible later. An iteration in GAs is also known as a generation. In step 2, the algorithm enters a loop from which it exits only if the stopping condition has been met. Typically, there are three conditions that could be used, which are one or a combination of the following [4]:

- The Individuals have achieved convergence. This means that the Population of a generation is not sufficiently different from the Population of the previous generation.
- A solution of acceptable quality has been found.
- Computation has exceeded the allotted time.

When the loop starts, the algorithm needs to select some Individuals; to achieve that, an evaluation method is required such that there is a distinction of the quality between Individuals. To evaluate the quality of a given Individual, the GA uses a Fitness Function (FF), which takes as input the vector that represents that Individual and has as an output a numerical value denoting how good this Individual is. For ARCWMs, there exist multiple

FFs, which were discussed in Section 4: accuracy, area, delay, and power. Since a typical GA requires that there is a single FF, a weighted average of the four FFs could be used. GA's thus perform optimization by finding the Individuals that produce the best outputs when given as input to the FF.

In step 3, the GA selects some of the Individuals in the Population to use them as combination parents; the better the fitness score of the Individual, the more likely it is to be selected. This is performed with a selection operator, which takes as input the Population and produces as output a subset of the Population containing the selected Individuals. An Individual might be selected more than once in this process, in which case both occurrences are counted. The simplest selection operator selects the best Individuals and uses them as the Population for the next generation. In practice, this method is not used, since it could lead the algorithm to become stuck in a local optimum area if all of the best Individuals are located there. To alleviate this, randomness needs to be introduced, such that we are choosing randomly, but the best Individuals have a higher chance of being selected. The most common selection operators are the Tournament Selector and Roulette Selector [4] (Figure 3).

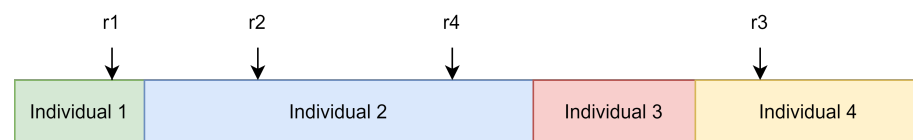


Figure 3. Example of roulette selection: four Individuals are placed on the roulette. Their area corresponds to their score. Four random numbers, $r1$ – $r4$, are chosen. Individual 1 is selected once, since $r1$ landed on its region; Individual 2 is selected twice by $r2$ and $r4$, and Individual 4 is selected once by $r3$.

The selected Individuals are combined in step 4 to produce new Individuals, which are added to the Population. Combination is a function that takes two Individuals as input and produces two new Individuals as output. To illustrate combination in an intuitive way, the input Individuals need to be considered as vectors, the element values or “genes” of which are combined into the genes of the output Individuals. Combination aims to take the best genes for each element from the two Parents and pass them on to at least one of the Children, producing an Individual with a higher score than the two parents (Figure 4). In the literature, it is common practice to convert values of each gene into binary numbers and apply the process above, not at each element index, but at each bit index instead; this is called Binary Combination [4]. Manipulations in binary form allow for genes to be sliced, and give the Children combined genes instead of passing them on intact.

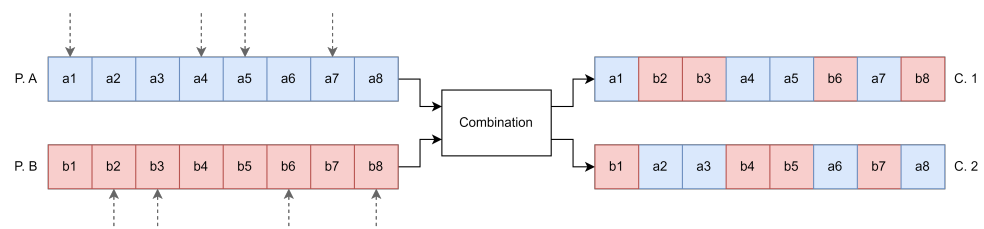


Figure 4. Numerical combination: In this example, the mask vector is (1,0,0,1,1,0,1,0), as shown by the light gray arrows. The first Child has the values of Parent A at the indexes where the mask has a 1 and the values of Parent B where the mask has a 0. Child 2 is the complementary of this process.

Additionally the GA, in step 5, randomly mutates some of the Individuals of the Population, which alters some part of their solution. Usually, as with Combination, the genes are converted into binary, and one or more of the gene's bits are flipped [4]. Mutation

allows the GA to escape from local optima produced by the same Individuals being repetitively selected. The newly added and mutated Individuals need to be evaluated so that each Individual in the Population has its fitness score. These new Individuals are how the GA navigates the search space, with the premise that if two Individuals are good solutions to the problem, their combination of solutions might produce an even better solution. Finally, in step 6, to keep the Population size constant, the Individuals with the lowest fitness scores are discarded. The above procedure is a generation of the GA, and it is repeated until a stopping criterion is met, which is usually a limit on the generation number or if an acceptably good value of fitness score is met by an Individual, making it the best solution.

Occasionally a GA's search space can be reduced by using variable constraints. This allows the GA to ignore Individuals that are not candidates for quality solutions. For this work, such constraints could take the form of not allowing a slot to be filled with an AFA, forcing it to be filled with an FA, or not utilizing a specific AFA, thus restricting part of the search space. If a designer chooses so, they can easily implement such constraints into this methodology. In this paper, no constraints were implemented for the values that can be assigned to the individual variables. This is because there is no clear way to find such constraints. This means that, for the general case, any Individual in the search space is a valid candidate to be a high quality ARCWM.

5.2. NSGA-II

As mentioned above, for the ARCWMs, there are multiple FFs to optimize. In order to be able to apply the GA approach with multiple objectives, the NSGA-II method [6] is chosen. NSGA-II operates on the same basis that GAs do, but it employs a different approach on sorting the Individuals within the Population based on their fitness. In a standard GA, it can be argued that an Individual is more fit than another if its Fitness Score is better. For NSGA-II, an Individual is more fit than another if it dominates the other. An Individual is considered to dominate another Individual if it is better than it in at least one objective, and is better than or equal at every other objective,

$$\exists o \in O : A.o \succ B.o \quad (5)$$

$$\forall o \in O : A.o \succeq B.o \quad (6)$$

where A and B are Individuals, o is an objective from the set of all objectives O , $A.o$ represents the value of objective o for Individual A , and $\succ$ and $\succeq$ denote that the left hand value is "better than" and "better than or equal to" the right hand value.

NSGA-II uses this notion of dominance in order to sort the Individuals in the Population into ranks of "non-dominance": all Individuals that are not dominated by any other Individual are given a rank of zero, all Individuals that are dominated by exactly one other Individual are given a rank of one, etc. All Individuals within a rank are considered equal by the algorithm. During each generation of the algorithm, the non-dominated sorting method is used in step 3 and 6, new Individuals are generated using the same Selection–Crossover–Mutation methods used by GA. In this way, elitism is promoted despite the multiple objectives.

After the doubling of the Population, in step 6, the non-dominated sorting takes place and the best Individuals are kept to advance to the next Population. When this happens Individuals within the highest non-dominance ranks are selected, while ones within the worst ranks are discarded. When the Population is culled one of the middle non-dominance ranks might be cut—i.e., the algorithm needs to keep only some of these Individuals and not the

entire rank, see Figure 5. Thus, there needs to be a way to distinguish between Individuals within the same non-dominance rank. NSGA-II achieves this using crowding distance.

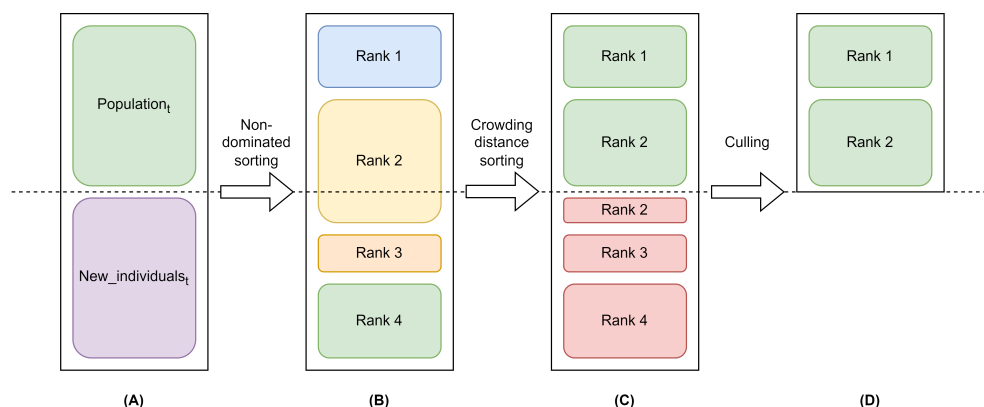


Figure 5. NSGA-II culling procedure example: (A) the Population during generation t is doubled, new Individuals are produced from the Population using Selection–Crossover–Mutation; (B) the total Population is sorted into four ranks using non-dominated sorting; (C) rank 2 is sorted using Crowding distance sorting; (D) only rank 1 and the top Individuals in rank 2 remain for the Population at the beginning of generation $t + 1$.

Crowding distance is a metric of how crowded an Individual's objective values are. For each objective, all of the Individuals within the same rank are sorted by their value (or score, in this objective). For one Individual, its crowding distance for that objective is equal to the absolute normalized difference in the objective values of its two adjacent Individuals. Normalization is achieved by dividing the absolute difference by the difference of the maximum and minimum limits of this objective, which are specified as parameters of the algorithm. Individuals at the boundaries of the objective value sorting (i.e., the very first and last), which have only one adjacent Individual, are given an infinite crowding distance for this objective. Once the crowding distance has been evaluated for every objective, each Individual's total crowding distance is the sum of its crowding distances for each objective. Between two Individuals within the same rank, the one with the highest crowding distance is considered better; this is performed to promote diversity in the solutions.

Using the methods of non-dominated sorting and crowding distance sorting, the NSGA-II manages to operate on multiple objectives and, at the same time, it generates solutions that are Pareto-optimal. For this work, the NSGA-II was used to optimize approximate multipliers, built following the ARCWM model, for the FFs of accuracy, area, delay, and power. The cases of 8×8 , 16×16 , and 32×32 ARCWMs were evolved for 100 generations each.

The flow of the entire methodology developed in Python can be seen in Figure 6. The diagram is separated into two sections: NSGA-II and evaluation method. NSGA-II is the main loop of the program, executed for 100 generations, which iteratively optimizes a Population of 100 Individuals. It takes as input a random initial Population, and returns as output the optimized Population. The evaluation method is used by NSGA-II every time there are new non-evaluated Individuals. It takes as input an Individual, evaluates it, and returns as output the FF values for it.

5.3. NSGA-II Parameters

Parameters for this work include a Population size of 100, alongside a Tournament Selector with 20 selected Individuals and a Tournament size of 2. Each generation the Population is doubled with the addition of 100 children, through one-point crossover and

1-bit mutation to the selected Individuals. The mutation probability is 0.07, resulting in an average of seven mutations per generation. The FFs used are

$$I.Accuracy = MAE(I) \quad (7)$$

$$I.Area = AREA(I) \quad (8)$$

$$I.Delay = CLK - SLACK(I) \quad (9)$$

$$I.Power = POWER(I) \cdot 1000 \quad (10)$$

where I is an Individual, $MAE(I)$ is the MAE (Equation (4)) as reported by the C++ functional simulation program mentioned in Section 4, $AREA(I)$ is the circuit size requirement of I as reported by Genus, CLK is a delay constraint given per multiplication size, $SLACK(I)$ is the delay constraint slack of I as reported by Genus, and $POWER$ is the circuit power consumption of I as reported by Genus. Meta-optimization of the NSGA-II was not studied in this work.

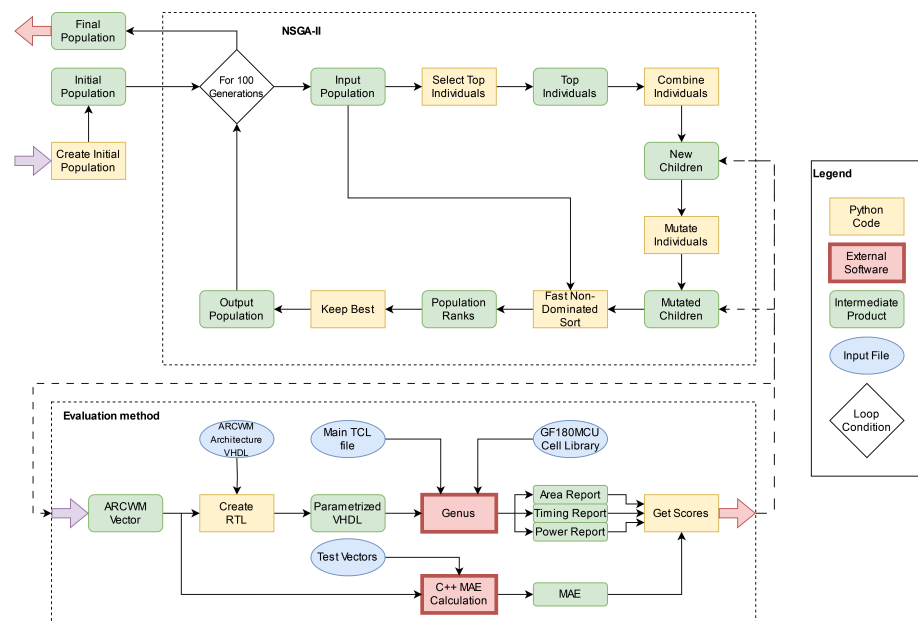


Figure 6. Flow of the ARCWM method. Big arrows denote input (purple arrow) and output (red arrow). Yellow squares are the main Python code, red squares are external software, blue ovals are input files, and green squares with rounded corners are intermediate products. Small arrows show the flow of data, from intermediate products to code and vice versa.

A delay constraint orders Genus to optimize the circuit until it is able to function properly with that delay as a clock period. When Genus finishes optimizing, it reports a slack. Slack is the time that the delay constraint can be reduced while the circuit still functions properly. If the slack is negative, that means that Genus failed to meet the constraint, and the delay needs to be increased by the absolute amount of slack for the circuit to function properly. CLK is set to the following values:

$$CLK = \begin{cases} 14 \text{ ns} & 8 \times 8 \text{ ARCWM} \\ 26 \text{ ns} & 16 \times 16 \text{ ARCWM} \\ 33 \text{ ns} & 32 \times 32 \text{ ARCWM} \end{cases} \quad (11)$$

Note that these are the values used for the manufacturing technology presented at Section 4, a designer should set these values close to the delay that the RCWM has when simulated with their manufacturing technology. Thus, $I.Delay$ shows the minimum clock

period that an ARCWM can operate on. Power is reported as W and is multiplied by 1000 to be converted to mW.

5.4. Computational Complexity Analysis

The NSGA-II algorithm's complexity is mainly dependent on the Population size [6]. That is because the complexity of partial order sort is $O(MN^2)$, where M is the number of FFs and N is the Population size. Since $M = 4$ and constant, partial order sort complexity can be considered $O(N^2)$. In one iteration of the algorithm, the selection, combination, and mutation methods are also executed. Selection is performed using the Tournament Selector, and has a complexity of $O(S \cdot (S + S \cdot \log(S)))$, where S is the amount of selected Individuals, essentially $O(S^2)$, but since $S = 2$ and constant, it can be considered $O(1)$ with regard to N . The combination's complexity is $O(C \cdot K)$, where C is the amount of Children produced and K is the amount of variables in an Individual; since $N = K$, the complexity of combination is $O(N \cdot K)$. The complexity of mutation is also $O(N \cdot K)$ in the worst case where every Individual mutates, but that is highly unlikely with a mutation probability of 0.07. Thus, the average complexity of mutation is $O(N)$. Finally, culling the Population and keeping the best N Individuals requires the crowding-distance assignment method, which has a complexity of $O(M \cdot N \cdot \log(N))$. All of the above procedures are run serially one after another (see Figure 6) thus the total complexity of the NSGA-II algorithm is $O(N^2 + N \cdot K)$. $K = 39$ for 8×8 ARCWMs, but for 16×16 ARCWMs, $K = 201$, and for 32×32 ARCWMs, $K = 907$; assuming that the designer wants to scale the Population size, the algorithm's complexity is dependent on N^2 or $N \cdot K$, based on the ARCWM's multiplication size.

The complexities discussed for combination and mutation methods do not include the complexity of evaluation of the FFs. After creating new Individuals with the combination and mutation methods, these Individuals need to be evaluated. Evaluation requires running the evaluation method (see Figure 6) for at most $2N$ new ARCWMs— N children and N mutations—. The complexity of evaluation is dependent on the complexity of the MAE calculation and the complexity of Genus Synthesis and Implementation. The complexity of the MAE calculation for one ARCWM is $O(T \cdot K)$ where T is the amount of test input pairs and K is the amount of variables in an Individual. T is limited to 10^7 to keep the run-time of this calculation low, as discussed in Section 4. It is impossible to analyze the complexity for Genus since it is a closed-source program so it is assigned a complexity of $O(G(V))$, where V is the VHDL input of an ARCWM and G is some function of V . Thus, the total complexity of the methodology, including the NSGA-II algorithm and evaluation of FF, is $O(N^2 + N \cdot K + N \cdot (G(V) + T \cdot K))$.

During testing, it was observed that $O(N \cdot (G(V) + T \cdot K)) > O(N^2 + N \cdot K)$. This means that a designer can approximate the required run-time for completion of a generation of the algorithm for their system by first calculating the evaluation times shown in Table 4, and then multiplying them with the Population size. This generation time can then be multiplied with the amount of generations to obtain an estimate of the total required run-time. This estimate will be higher than the actual required time because in later generations the Population includes repeat Individuals, thus the total amount of Individuals for evaluation is slightly smaller than N .

6. Experimental Results

Using our NSGA-II-based methodology, the cases of 8×8 , 16×16 and 32×32 ARCWMs were optimized. The algorithm outputs a Pareto optimal set of solutions optimized for accuracy, area, delay, and power. This output data are in JavaScript Object Notation (JSON) format for each Individual in the set, and contains an integer vector denoting, where AFAs fill each slot of the ARCWM, and the evaluated accuracy, area, delay, and power. In this

section, we present a statistical analysis of these data (available at [9]) to demonstrate the effectiveness of the methodology.

In Tables 5–7, the minimum and maximum reductions are shown as a percentage of the RCWM, meaning that, in 8×8 ARCWMs, in the best case, the area is reduced by 66.3% of the RCWM (i.e., the best 8×8 ARCWM's area is equal to 33.7% of the 8×8 RCWM's area). As shown in these tables, area reduction is good for both the best and worst cases in 16×16 and 32×32 ARCWMs; for 8×8 ARCWMs, the area reduction is good only for the best case. Delay reduction is sub-par for 8×8 ARCWMs, and virtually non-existent for the 16×16 and 32×32 cases. As explained in Section 4, this is due to the setting of a clock constraint, upon which the synthesis optimization of ARCWMs stops. Power, similarly to area, but slightly better overall, obtains an adequate reduction in 16×16 and 32×32 ARCWMs, reaching up to a 70% power reduction, with a smaller achievement in 8×8 ARCWMs.

These tables do not contain accuracy, as accuracy cannot be directly compared with the RCWM. The RCWM has zero MAE and, thus, there cannot be a measure of how much more inaccurate an approximate circuit is compared with its exact counterpart. Comparisons can only be performed in between inaccurate circuits of the same type.

Table 5. Minimum–maximum reduction vs. RCWM, for 8×8 ARCWMs.

	Area (%)	Delay (%)	Power (%)
Max Reduction	66.3	26.2	57.9
Min Reduction	34.5	19.4	33.8

Table 6. Minimum–maximum reduction vs. RCWM, for 16×16 ARCWMs.

	Area (%)	Delay (%)	Power (%)
Max Reduction	65.9	1.4	70.7
Min Reduction	60.8	1.0	63.6

Table 7. Minimum–maximum reduction vs. RCWM, for 32×32 ARCWMs.

	Area (%)	Delay (%)	Power (%)
Max Reduction	63.4	1.1	69.0
Min Reduction	58.0	1.0	6.1

For each case, the RCWM is evaluated, and then the produced ARCWM Pareto front is compared with it. The statistics for the ARCWM Pareto fronts are displayed in Table 8 for 8×8 ARCWMs, Table 9 for 16×16 ARCWMs, and Table 10 for 32×32 ARCWMs. In these tables, the mean, standard deviation, minimum, maximum, and RCWM values are given for the characteristics of accuracy, area, delay, and power. These same results are visualized in Figures 7–9 for 8×8 , 16×16 , and 32×32 , respectively. Each figure contains three graphs, and each graph plots accuracy (MAE) against area, delay, and power, one per graph. This way, the relationship between accuracy and the physical characteristics of the ARCWMs (blue crosses) in the Pareto front can be intuitively shown.

In these graphs, the curve formed by the data points displays the following behavior: when accuracy becomes worse, the other physical characteristic becomes better. For example, in the MAE–area (Figure 7A) graph it is anticipated that an 8×8 ARCWM with smaller area is less accurate. Thus, it can be hypothesized that as ARCWMs begin in the optimization process in random configuration, the curve of the Pareto front is at the top right of the graph (high MAE and physical characteristics). As the ARCWMs become more

and more optimized through the generations, this curve is pushed towards zero (bottom left corner). In the graphs, the optimal front for this physical characteristic is also shown (red line). This line crosses the points of only the optimal ARCWMs for each physical characteristic (i.e., as accuracy worsens, an ARCWM is part of the optimal front only if it improves the physical characteristics). Finally, to show the improvement in the physical characteristics compared with the RCWM, it is plotted as a star.

Table 8. Statistics for 8×8 ARCWMs.

(Samples = 88)	Accuracy (MAE)	Area (μm^2)	Delay (ps)	Power (mW)
mean	2782.113636	6625.861977	14,084.068182	0.306212
std	459.108542	1438.739196	224.158493	0.044477
min	2471.000000	5016.032000	13,909.000000	0.251661
max	5386.000000	9744.493000	15,197.000000	0.396161
norm	0.000000	14,872.480000	18,857.000000	0.598125

Table 9. Statistics for 16×16 ARCWMs.

(Samples = 95)	Accuracy (MAE)	Area (μm^2)	Delay (ps)	Power (mW)
mean	2.109085×10^8	19,577.995253	25,985.168421	0.424213
std	1.041313×10^8	578.191736	15.994449	0.021105
min	1.296941×10^8	18,286.016000	25,912.000000	0.380772
max	4.395964×10^8	21,003.674000	26,000.000000	0.474001
norm	0.000000×10^0	53,619.955000	26,273.000000	1.301030

Table 10. Statistics for 32×32 ARCWMs.

(Samples = 94)	Accuracy (MAE)	Area (μm^2)	Delay (ps)	Power (mW)
mean	7.421317×10^{17}	74,515.830543	32,997.042553	1.318739
std	1.812024×10^{17}	3223.052175	3.185704	0.092922
min	5.894236×10^{17}	70,101.517000	32,984.000000	1.175320
max	1.522444×10^{18}	80,537.498000	33,000.000000	1.481320
norm	0.000000×10^0	191,715.597000	33,341.000000	3.793500

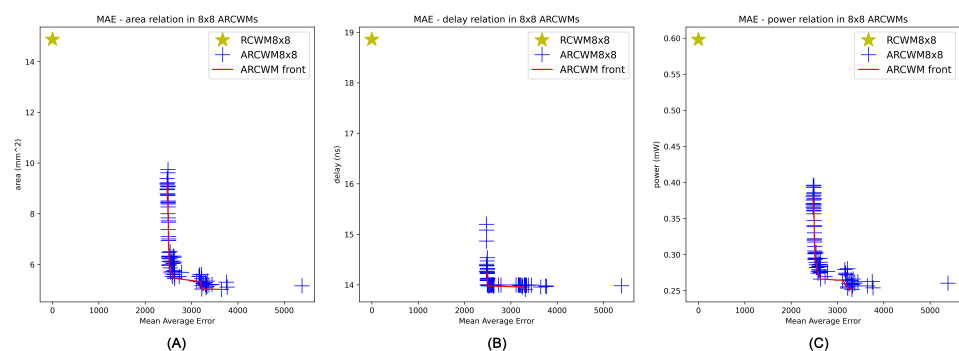


Figure 7. Plots of the relation of MAE with (A) area, (B) delay, and (C) power for 8×8 ARCWMs. The RCWM's characteristics are shown with a yellow star, its MAE is always zero. Blue crosses show the top 88 ARCWMs, while the red line shows the Pareto front for each objective.

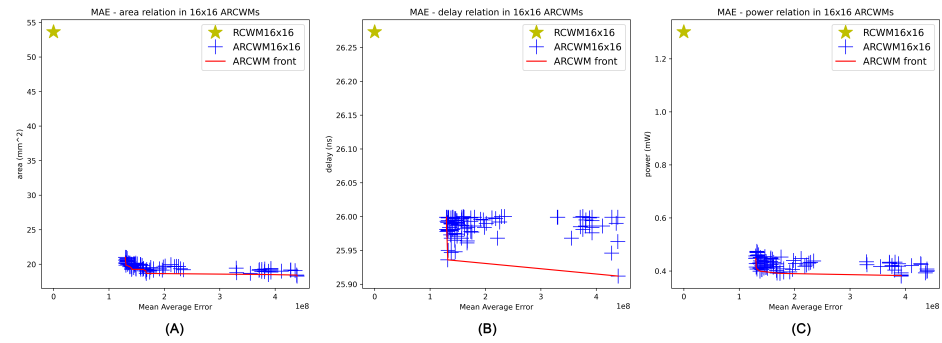


Figure 8. Plots of the relation of MAE with (A) area, (B) delay, and (C) power for 16×16 ARCWMs. The RCWM's characteristics are shown with a yellow star, its MAE is always zero. Blue crosses show the top 95 ARCWMs, while the red line shows the Pareto front for each objective.

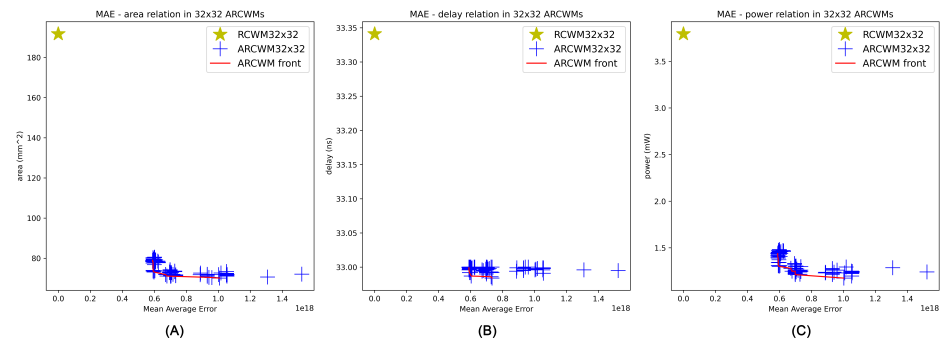


Figure 9. Plots of the relation of MAE with (A) area, (B) delay, and (C) power for 32×32 ARCWMs. The RCWM's characteristics are shown with a yellow star, its MAE is always zero. Blue crosses show the top 94 ARCWMs, while the red line shows the Pareto front for each objective.

ARCWM Scaling

As the amount of gates reported in Water's and Swartzlander's work [3] is increasing exponentially for the different multiplication sizes of the exact RCWM, the same is true for the resulting area after synthesis (Figure 10). Delay seems to scale logarithmically; this is similar to how the stage amount scales logarithmically in [3]. The critical path of this circuit can be found at the route one of the partial products will follow down the center of the multiplier to one of the middle bits of the output. Consequently, the delay is correlated with the amount of stages an ARCWM has and, thus, its multiplication size. Power scales exponentially similarly with area. A designer could predict how an RCWM will scale based on its multiplication size and the logarithmic and exponential curves fitted to the data from synthesis and implementation of the 8×8 , 16×16 , and 32×32 provided in this work.

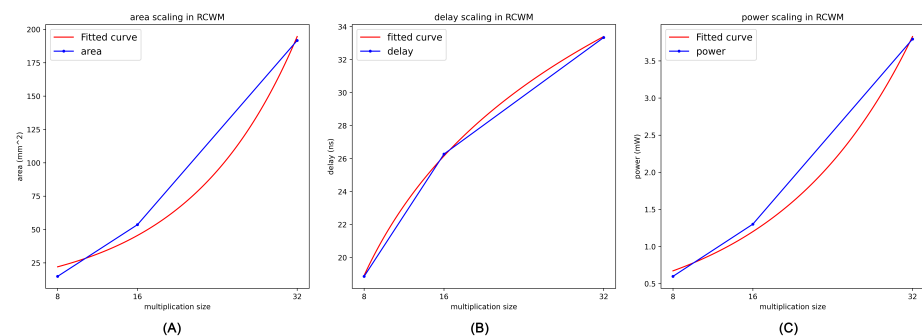


Figure 10. Scaling graphs for synthesized circuit characteristics: on the x-axis, the graph displays multiplication size (8×8 , 16×16 and 32×32) of the y-axis values for the corresponding physical characteristic, (A) area, (B) delay, and (C) power, are shown. Also, the corresponding exponential or logarithmic fitted curves are drawn using the Least Squares Method.

7. Related Work

As noted in [13], the wide range of approximate multipliers available in the literature can be classified based on the approximation techniques used. These techniques include

1. Truncation and rounding;
2. Approximate radix encodings;
3. The use of approximate compressors, and;
4. Logarithmic approximation.

In this work, the proposed method for creating an ARCWM relies on the utilization of approximate compressors, specifically the use of approximate Full Adders (3-to-2 compressors), developed via CFA by modifying the Adder's truth table. As mentioned above, the literature also provides some open-source libraries of approximate arithmetic circuits, such as SMApplLib [23] and EvoApprox8b [24].

SMApplLib [23] makes use of the underlying structure of the Look Up Tables (LUTs) and carry chains of modern FPGAs to develop a methodology for designing approximate multipliers specifically optimized for FPGA-based systems. The authors propose a design space exploration methodology for generating approximate multipliers of arbitrary data sizes. For each $N \times N$ accurate multiplier, three approximate $N \times N$ multiplier designs are offered, leveraging efficient utilization of LUTs and carry chains. To accelerate the execution time of an $N \times N$ multiplier, the methodology suggests implementing it using four instances of $N/2$ multipliers. Each $N/2$ multiplier can be generated either directly or recursively from four instances of $N/4$ multipliers. Additionally, the approach introduces a novel N -bit approximate Adder to reduce the overall execution time while supporting accurate summation of partial products. We note that, in [23], there is no specific mention of an optimization method for exploring the search space. From the results, we can assume that a multi-objective method, likely NSGA-II, is used, but this statement can not be confirmed from the text. The approximate multipliers are provided in the dataset [27].

In [24], the automated design of approximate circuits with formal error guarantees is discussed. The authors introduce EvoApprox8b, which is a comprehensive repository of approximate Adders and multipliers, with the primary objective of establishing a standardized framework for evaluating approximation techniques for these circuits. The circuits were produced using the multi-objective genetic algorithm NSGA-II [6] with a Cartesian Genetic Programming (CGP) [28] representation for the circuits. In [29], the authors of EvoApprox8b use the CGP&NSGA-II method for 32×32 approximate multipliers. Additionally, they introduce a novel approach of computing the total Worst Case Absolute Error (WCAE) for these multipliers. The WCAE denotes the maximum error achievable across all possible outputs by an approximate multiplier. WCAE is used to streamline fitness function computations, avoiding the need for further evaluation of a multiplier if the error fails to surpass the current best. Our approximation methodology differs from the EvoApprox8b's CGP approach in this regard: while EvoApprox8b simplifies multipliers by modifying their internal wiring and eliminating unused circuit components, our approach preserves the structure of the reduced-complexity Wallace multiplier and substitutes FAs with AFAs.

In Table 11, we give a high level comparison of all three works on approximate multiplier design using GAs. The comparison characteristics are the application platform for which this method is tailored, the multiplication sizes that can be achieved with this methodology, the type of model used, the search algorithm used, and a link to the dataset. The type of model used means the basis on which the approximate multiplier is designed. In our case, an RCWM was modified; for EvoApproxLib, CGP was used, which creates an arbitrary circuit; and for SMApplLib, the Wallace tree multiplier was built using LUTs as the basic building blocks.

Table 11. Comparison table for works related to ARCWM.

Methodology	Application Platform	Multiplication Sizes Provided	Model	Search Algorithm	Dataset/Tool Flow Available at
ARCWM	ASIC	8×8 , 16×16 , 32×32	RCWM [3] with AFAs	NSGA-II [6]	[9]
EvoApproxLib [24]	ASIC	8×8 , 16×16 , 32×32	CGP [28]	NSGA-II [6]	[30]
SMApoxLib [23]	FGPA	Any size $N \times N$	Wallace tree multiplier [2] built with LUTs	Unspecified	[27]

8. Discussion

In their work on the EvoApprox8b library [24], Mrazek et al. point out the following qualities that papers dealing with circuit approximation should show:

1. A corresponding software implementation of the approximation method should be available.
2. An implementation of the original (exact) circuit should be available.
3. Results such as the quality of approximation and other parameters of approximate circuits should be easily reproducible.
4. Implementations of the resulting approximate circuits should be available.
5. A variety of approximate versions created from the original circuit should be reported, thus forming a densely occupied Pareto front.
6. It should be made clear if the given test vectors used to evaluate approximate circuits are sufficient for obtaining a trustworthy error quantification.
7. The given approximation method should be compared against competitive approximation methods.

Mrazek et al. comment that papers on circuit approximation rarely follow these qualities in their entirety. Qualities 1 and 3 are somewhat equivalent, because reproducibility should be possible without making it necessary for a designer to have to rewrite the software used to produce the multipliers. Of course, reproducibility is a requirement for a high-quality work. Quality no. 5 is understandable, as a dense Pareto front reveals the quality of the optimization and generally the more the data under discussion the better.

In our prior work on 32×32 ARCWMs [8], we compare our 32×32 ARCWMs with 32×32 approximate multipliers from the EvoApprox8b library. We note that, at some time between the release of that paper and the writing of this work, the data and source files used were made private by Mrazek et al., and thus are no longer available for download. This means that, as of the writing of this paper, there are no available data within the EvoApprox8b library for comparison of 32×32 approximate multipliers that we are aware of. The most updated version of the EvoApproxLib website [30] contains only data on 8×8 and 16×16 approximate unsigned multipliers and other circuits that are unrelated to this work. When comparing with 32×32 EvoApprox8b multipliers, they needed to be synthesized and implemented for the manufacturing technology we utilized in our evaluation methodology. Instead of synthesizing the multipliers, Mrazek et al. used an estimate for area, delay, and power consumption [31]. There is no easy way to directly translate the available data on the physical characteristics of the EvoApprox8b multipliers produced by one evaluation methodology to another. Thus, it is necessary when making a comparison that circuits from all sources are synthesized using the same methodology and raw results are of no use in this effort. This highlights why qualities 2 and 4 described above are required.

Furthermore, on the topic of comparison, and quality no. 6 and 7, we would like to bring forth the argument of application specificity in the discussion of approximate circuits. As mentioned numerous times throughout this work, optimization of approximate multipliers can be performed utilizing knowledge of the application that these are going to be used in, and this extends to most types of approximate circuits. By knowing the characteristics of the application, the designer can tune the parameters of the optimization algorithm such that they obtain results that are more performant for that case. We show that in the comparison we perform for 32×32 ARCWMs [8], the MAE for our ARCWM is better because the NSGA-II algorithm optimized them with this objective, while EvoApprox8b multipliers were optimized using Mean Relative Error (MRE).

When researching a method such as ARCWM or CGP, we are limited to attempting to evaluate the general case, agnostic to a specific application's requirements. While a comparison against competitive methods might sound like a reasonable course of action for a research work on approximation methods, we need to note that this comparison is performed for the general case. We believe that providing the designers with as many tools and insights into a method is much more valuable in their endeavor of choosing an approximation method for their application than a raw general comparison.

Furthermore, more equal grounds for comparison need to be established to facilitate fair comparison of the general case. That is a convention on a set of test vectors, for cases where fully evaluating with all possible inputs is impractical, a convention on the synthesis methodology and technology and, perhaps, a convention on some test applications to be used for testing the accuracy performance of the approximate circuits in simulation. Development of an evaluation pipeline, using open source tools and technology libraries, could greatly help researchers with evaluation and comparison of their approximation methods. For the time being, as such a convention does not exist, we want to emphasize that knowledge of the details of the evaluation methodology, as well as the source code by which it was achieved, is just as important for comparisons as the source code of the circuits.

To the best of our knowledge, as of the writing of this paper, EvoApproxLib [30] and SMapproxLib [27] are the only openly available datasets that offer Register-Transfer Level (RTL) code for approximate multipliers. SMapproxLib is designed for FPGAs and, as such, it would not be a good practice to compare it with ASIC based approaches. As area, delay, and power are related to LUT usage within FPGAs rather than physical cells, the optimization goals are vastly different when an FPGA is used. For EvoApproxLib, only about 30 8×8 and 16×16 multipliers are available; no 32×32 multipliers are available. As we mentioned before, the RTL code is necessary in order to have the EvoApproxLib multipliers synthesized with our own synthesis methodology. The C files are also available, which allow for the calculation of the MAE for the EvoApproxLib multipliers. Since the set of test vectors used to calculate MAE for EvoApproxLib might differ from the ones we used for ARCWMs, and we only have the latter set and have optimized for it, we will not be comparing accuracy. We can still compare area, delay, and power, as seen in Figure 11.

Figure 11 allows for comparison of the multipliers from both studies with the RCWM, as it can be seen in terms of area all multipliers from both studies have a reduction better than 50%. For delay and power, some EvoApproxLib multipliers perform worse than the RCWM, whereas ARCWMs are guaranteed to perform at least equal to the RCWM; the equal case happens in delay which is explained in Section 6. Comparing ARCWMs with EvoApproxLib multipliers, we can see that some EvoApproxLib multipliers have better area, which is expected, since CGP has the ability to remove big chunks of the circuit, while ARCWMs replace FAs with AFAs, removing less at once. However, ARCWMs perform better in delay and power in most cases.

The results of this comparison could be misleading. That is because, first of all, the dataset for EvoApproxLib is too small (about 30, as of writing of this paper). Secondly, the optimization goals and evaluation methodology are different for ARCWMs and EvoApproxLib multipliers, but the only ones available to us are ours. With just these results as guide, it seems that this set of EvoApproxLib multipliers is more biased towards optimizing area than delay and power and, thus, might not be the best representatives of the EvoApproxLib 16×16 Pareto front. These might not be the data intended for comparison, but they are the only data available at this time.

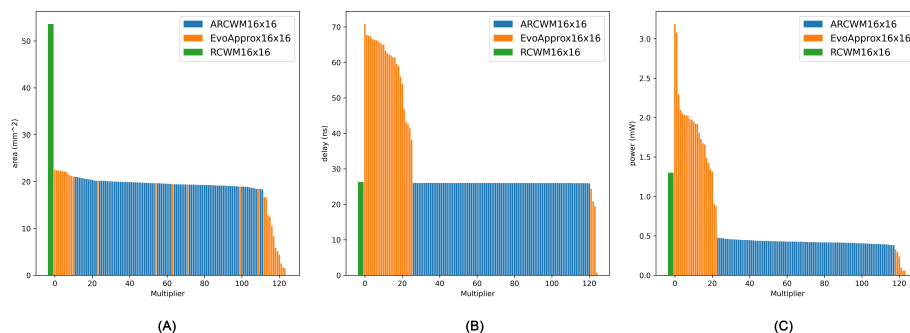


Figure 11. Bar plots of the (A) area, (B) delay, and (C) power comparisons between 16×16 EvoApproxLib and ARCWMs. The RCWM's characteristics are shown with a green bar. Blue bars show the ARCWMs, while the orange bars show the EvoApproxLib multipliers. The bars, except the RCWM bar, have been sorted by decreasing value (worst to best).

9. Conclusions

In this work, we developed the Approximate Reduced Complexity Wallace Multiplier by incorporating approximate Adders into the partial product reduction tree of the Reduced Complexity Wallace Multiplier. This approach was systematically optimized using the Non-Dominated Sorting Genetic Algorithm-II, enabling the simultaneous optimization of key design characteristics: accuracy, area, delay, and power consumption.

Using our methodology, the 8×8 , 16×16 , and 32×32 ARCWMs were optimized. The results demonstrate significant area and power reductions, particularly for the 16×16 and 32×32 ARCWMs, with area reductions reaching 66.3% of the RCWM in the best case for 8×8 ARCWMs. Delay reduction is minimal by design, due to the utilization of clock constraints during synthesis optimization. Nonetheless, power reduction reached up to 70% in 32×32 ARCWMs, showing promising trade-offs between accuracy and physical characteristics.

When comparing ARCWMs with approximate multipliers, such as those from the EvoApproxLib library, limitations arise due to differences in evaluation methodologies. This highlights the importance of standardizing evaluation pipelines, including synthesis methodologies, test vectors, and simulation frameworks, to ensure fair comparisons between approximation methods for the general case, whereas when an approximate arithmetic circuit is designed for a specific application, its comparison becomes difficult to determine. We argue that providing designers with comprehensive tools, insights, and open-source resources for ARCWM optimization is more valuable than raw general comparisons. To this end, we have released the VHDL code and associated witholds as open-source resources [9], enabling researchers and designers to explore, evaluate, and adopt the ARCWM methodology for their specific needs. The ARCWM architecture offers a highly customizable and efficient solution for approximate multipliers, balancing trade-offs between accuracy and physical characteristics. Future work will explore ARCWM implementations in application-specific domains such as machine learning accelerators and signal processing systems, where approximate computing demonstrates significant potential.

Author Contributions: Conceptualization, I.R., G.P. and A.E.; methodology, I.R. and G.P.; software, I.R. and G.P.; validation, I.R.; formal analysis, I.R.; investigation, I.R.; resources, A.E.; data curation, I.R.; writing—original draft preparation, I.R., G.P. and A.E.; writing—review and editing, I.R., G.P. and A.E.; visualization, I.R. and G.P.; supervision, A.E.; project administration, A.E. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All data generated during this study are available at <https://github.com/jrizxos/Approximate-Reduced-Complexity-Wallace-Multipliers> (accessed on 9 December 2024). The EvoApproxLib LITE dataset is available at <https://ehw.fit.vutbr.cz/evoapproxlib/> (accessed on 9 December 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AC	Approximate Computing
AFA	Approximate Full Adder
ANS	Approximate Number System
ASIC	Application-Specific Integrated Circuit
ARCWM	Approximate Reduced Complexity Wallace Multipliers
CFA	Circuit Functional Approximation
CGP	Cartesian Genetic Programming
FA	Full Adder
FF	Fitness Function
FPGA	Field-Programmable Gate Arrays
GA	Genetic Algorithm
HA	Half Adder
JSON	JavaScript Object Notation
LUT	Look Up Table
MAE	Mean Absolute Error
MED	Mean Error Distance
MRE	Mean Relative Error
OC	Over-Clocking
NSGA-II	Non-Dominated Multi-Objective Genetic Algorithm-II
RCWM	Reduced Complexity Wallace Multiplier
RTL	Register-Transfer Level
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VOS	Voltage Over-Scaling
WCAE	Worst Case Absolute Error

References

1. Rebooting the IT Revolution, a Call For Action. Technical Report, Semiconductor Industries Association, Semiconductor Research Corporation. 2015. Available online: <https://www.src.org/newsroom/rebooting-the-it-revolution.pdf> (accessed on 9 December 2024).
2. Wallace, C.S. A suggestion for a fast multiplier. *IEEE Trans. Electron. Comput.* **1964**, EC-13, 14–17.
3. Waters, R.S.; Swartzlander, E.E. A reduced complexity wallace multiplier reduction. *IEEE Trans. Comput.* **2010**, 59, 1134–1137. [CrossRef]
4. Marinakis, I.; Marinaki, M.; Matsatsinis, N.; Zopounidis, K. *Metaheuristic and Evolutionary Algorithms in Management Science Problems*; Kleidarithmos: Athens, Greece, 2011.
5. Rovithakis, G. *Optimization Techniques*; Tziola Publications: Thessaloniki, Greece, 2020.

6. Deb, K.; Pratap, A.; Agarwal, S.; Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* **2002**, *6*, 182–197. [\[CrossRef\]](#)
7. Papatheodorou, G.; Rizos, I.; Efthymiou, A. Design Space Exploration of Partial Product Reduction Stage on 8x8 Approximate Multipliers. In Proceedings of the 2024 Panhellenic Conference on Electronics & Telecommunications (PACET), Thessaloniki, Greece, 28–29 March 2024; pp. 1–4.
8. Rizos, I.; Papatheodorou, G.; Efthymiou, A. Exploring the Design Space of 32x32 Approximate Reduced Complexity Wallace Multipliers. In Proceedings of the 2024 13th International Conference on Modern Circuits and Systems Technologies (MOCASST), Sofia, Bulgaria, 26–28 June 2024; pp. 1–4.
9. Approximate Reduced Complexity Wallace Multipliers. 2024. Available online: <https://github.com/jrizxos/Approximate-Reduced-Complexity-Wallace-Multipliers> (accessed on 9 December 2024).
10. Dennard, R.; Gaensslen, F.; Yu, H.N.; Rideout, V.; Bassous, E.; LeBlanc, A. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circuits* **1974**, *9*, 256–268. [\[CrossRef\]](#)
11. Moore, G.E. Cramming more components onto integrated circuits. *Proc. IEEE* **1998**, *86*, 82–85. [\[CrossRef\]](#)
12. Hennessy, J.L.; Patterson, D.A. A new golden age for computer architecture. *Commun. ACM* **2019**, *62*, 48–60. [\[CrossRef\]](#)
13. Leon, V.; Hanif, M.A.; Armeniakos, G.; Jiao, X.; Shafique, M.; Pekmestzi, K.; Soudris, D. Approximate computing survey, Part I: Terminology and software & hardware approximation techniques. *arXiv* **2023**, arXiv:2307.11124.
14. Carbin, M.; Kim, D.; Misailovic, S.; Rinard, M.C. Proving acceptability properties of relaxed nondeterministic approximate programs. *ACM SIGPLAN Not.* **2012**, *47*, 169–180. [\[CrossRef\]](#)
15. Chakradhar, S.T.; Raghunathan, A. Best-effort computing: Re-thinking parallel software and hardware. In Proceedings of the 47th Design Automation Conference, Anaheim, CA, USA, 13–18 June 2010; pp. 865–870.
16. Chippa, V.K.; Mohapatra, D.; Roy, K.; Chakradhar, S.T.; Raghunathan, A. Scalable effort hardware design. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2014**, *22*, 2004–2016. [\[CrossRef\]](#)
17. Mittal, S. A survey of techniques for approximate computing. *ACM Comput. Surv. (CSUR)* **2016**, *48*, 1–33. [\[CrossRef\]](#)
18. Han, J.; Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In Proceedings of the 2013 18th IEEE European Test Symposium (ETS), Avignon, France, 27–30 May 2013; pp. 1–6.
19. Sampson, A. Hardware and software for approximate computing. Ph.D. Thesis, University of Washington, Seattle, WA, USA, 2015.
20. Laughlin, S.B.; Sejnowski, T.J. Communication in neuronal networks. *Science* **2003**, *301*, 1870–1874. [\[CrossRef\]](#) [\[PubMed\]](#)
21. Dehaene, S. The calculating brain. In *Mind, Brain, & Education: Neuroscience Implications for the Classroom*; Solution Tree Press: Bloomington, IN, USA, 2010; pp. 179–198.
22. Shafique, M.; Ahmad, W.; Hafiz, R.; Henkel, J. A low latency generic accuracy configurable adder. In Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, 8–12 June 2015; pp. 1–6.
23. Ullah, S.; Murthy, S.S.; Kumar, A. SMApplib: Library of FPGA-based approximate multipliers. In Proceedings of the 55th Annual Design Automation Conference, San Francisco, CA, USA, 24–28 June 2018; pp. 1–6.
24. Mrazek, V.; Hrbacek, R.; Vasicek, Z.; Sekanina, L. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 258–261.
25. Gowdar, C.V.; Parameshwara, M.; Sonoli, S. Comparative analysis of various approximate full adders under RTL codes. *ICTACT J. Microelectron.* **2020**, *6*, 947–952.
26. GlobalFoundries GF180MCU Open Source PDK. 2022. Available online: <https://github.com/google/gf180mcu-pdk> (accessed on 9 December 2024).
27. Tools and Downloads—cfaed. 2018. Available online: <https://cfaed.tu-dresden.de/pd-downloads> (accessed on 9 December 2024).
28. Miller, J.; Turner, A. Cartesian genetic programming. In Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, Madrid, Spain, 11–15 July 2015; pp. 179–198.
29. Češka, M.; Matyáš, J.; Mrazek, V.; Sekanina, L.; Vasicek, Z.; Vojnar, T. Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; pp. 416–423.
30. EvoApproxLib LITE The Basic Library of Approximate Circuits. 2022. Available online: <https://ehw.fit.vutbr.cz/evoapproxlib/> (accessed on 9 December 2024).
31. Hrbacek, R.; Mrazek, V.; Vasicek, Z. Automatic design of approximate circuits by means of multi-objective evolutionary algorithms. In Proceedings of the 2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), Istanbul, Turkey, 12–14 April 2016; pp. 1–6.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.