Fixed-point Encoding and Architecture Exploration for Residue Number Systems

BOBIN DENG, Computer Science, Kennesaw State University, Marietta, United States
BHARGAVA NADENDLA, Computer Science, Kennesaw State University, Marietta, United States
KUN SUO, Computer Science, Kennesaw State University, Marietta, United States
YIXIN XIE, Information Technology, Kennesaw State University, Marietta, United States
DAN CHIA-TIEN LO, Computer Science, Kennesaw State University, Marietta, United States

Residue Number Systems (RNS) demonstrate the fascinating potential to serve integer addition/multiplication-intensive applications. The complexity of Artificial Intelligence (AI) models has grown enormously in recent years. From a computer system's perspective, ensuring the training of these large-scale AI models within an adequate time and energy consumption has become a big concern. Matrix multiplication is a dominant subroutine in many prevailing AI models, with an addition/multiplication-intensive attribute. However, the data type of matrix multiplication within machine learning training typically requires real numbers, which indicates that RNS benefits for integer applications cannot be directly gained by AI training. The state-of-the-art RNS real number encodings, including floating-point and fixed-point, have defects and can be further enhanced. To transform default RNS benefits to the efficiency of large-scale AI training, we propose a low-cost and high-accuracy RNS fixed-point representation: *Single RNS Logical Partition (S-RNS-Logic-P) representation with Scaling Down Postprocessing Multiplication (SD-Post-Mul)*. Moreover, we extend the implementation details of the other two RNS fixed-point methods: *Double RNS Concatenation (D-RNS-Concat)* and *Single RNS Logical Partition (S-RNS-Logic-P) representation with Scaling Down Preprocessing Multiplication (SD-Pre-Mul)*. We also design the architectures of these three fixed-point multipliers. In empirical experiments, our *S-RNS-Logic-P representation with SD-Post-Mul* method achieves less latency and energy overhead while maintaining good accuracy. Furthermore, this method can easily extend to the Redundant Residue Number System (RRNS) to raise the efficiency of error-tolerant domains, such as improving the error correction efficiency of quantum computing.

CCS Concepts: • **Computer systems organization** → *Architectures*; • **Hardware** → Power and energy; • **Computing methodologies**; • **Mathematics of computing**;

Additional Key Words and Phrases: Residue Number System, RNS, Fixed-point Encoding and Computation, RNS Multiplier Architectures

1 INTRODUCTION

The Residue Number System (RNS) is a data encoding strategy particularly applied in domains such as DSP [9, 19], cryptography [10, 37], bioinformatics [14, 52], machine learning [50, 60], etc. The underlying principle of RNS encoding is to utilize a group of smaller bit-width numbers (a.k.a. residues) to stand for a larger bit-width value. E.g.,

Authors' Contact Information: Bobin Deng, Computer Science, Kennesaw State University, Marietta, Georgia, United States; e-mail: bdeng2@kennesaw.edu; Bhargava Nadendla, Computer Science, Kennesaw State University, Marietta, Georgia, United States; e-mail: bnadendl@students.kennesaw.edu; Kun Suo, Computer Science, Kennesaw State University, Marietta, Georgia, United States; e-mail: ksuo@kennesaw.edu; Yixin Xie, Information Technology, Kennesaw State University, Marietta, Georgia, United States; e-mail: yxie11@kennesaw.edu; Dan Chia-Tien Lo, Computer Science, Kennesaw State University, Marietta, Georgia, United States; e-mail: dlo2@kennesaw.edu.

ACM 1544-3973/2024/5-ART
https://doi.org/10.1145/3664923

four 8-bit integers may uniquely identify a 31-bit or 32-bit integer. Within a valid RNS data range, each RNS number (constructed by a group of residues) is bijectively mapped to a regular binary/decimal integer. For some arithmetical operations (addition, subtraction, and multiplication), the associated residues of different RNS operands are processed in parallel and carry-free with neighbors. So RNS acquires a certain degree of bit-level parallelism compared to conventional binary integer computation. Moreover, from the system architecture standpoint, using a set of smaller bit-width integer adders and multipliers to replace the default larger bit-width logic units should reduce the processor's execution latency, dynamic power, and area overhead. Essentially, not only integer applications but some arithmetic-intensive real number applications may also benefit from switching to RNS. E.g., matrix multiplications in CNN inference/training require real numbers and are addition/multiplication-intensive. One compelling benefit of RNS is its low-cost integer multiplier [56, 57], which guarantees that RNS becomes a competitive candidate for multiplication-intensive integer workloads. To retain the low power and latency rewards from RNS, how to represent real numbers in RNS while ensuring the efficiency of majority arithmetic operations is a crucial problem we should consider.

The AI models have evolved dramatically in recent years. E.g., the size growth of NLP models reaches 240x every two years [30]. The statistic from OpenAI [22] revealed that the training complexity of AI models doubles every 3.4 months, noticeably faster than the growth ratio of Moore's law (24 months). The worse situation is that Moore's law is predicted to end in 2033 and enter the post-Moore era [12]. Training large-scale AI models may take several months, even on supercomputers. E.g., the estimated training time of GPT4 [7, 61] is around 4-7 months [48]. Besides the high-performance requirement, low power should be another necessary metric when exploring next-generation supercomputers. The power cap guideline of an exascale supercomputer from DOE and DARPA is 20MW [27]. The first exascale supercomputer Frontier [17, 58] is 22.7 MW [5], close to the proposed limit. However, for the next-generation zettascale supercomputer (1000 exaflops/s), we may only have an extremely limited power budget to grow due to the restrictions of the cooling system and power supply constraints. So we have to delve into low-power strategies while substantially raising system performance. RNS is a competent candidate for low power and less latency in some specific applications. E.g., we can integrate some RNS-based hardware accelerators to serve addition/multiplication-intensive applications, such as matrix multiplications within CNN inference/training. These RNS-based AI accelerators could be installed in the next-generation extremely heterogeneous HPC to obtain lower power and fewer latency rewards. Besides the supercomputer, RNS-based architectures should also be extended to the edge/IoT domain because these devices typically have a very limited power budget. One potential application is Federated Learning [49] on edge/IoT.

The IEEE 754 Floating-point standard [1, 3] is extensively used in most binary systems to describe real numbers. Like the IEEE 754 standard, an RNS floating-point representation could be derived in which two distinct RNS integers represent the exponent and mantissa, respectively. The arithmetic operations of RNS floating-point are typically recognized as inefficient. Chiang etc. [39] proposed relatively low-cost RNS floating-point encoding and corresponding arithmetic algorithms. This methodology requires specific requirements, including all RNS moduli must be odd and coprime, and each RNS value needs to maintain a redundant residue to identify parity, etc. However, even with the above-specialized supports, the cost of arithmetical computations with RNS floating-point encoding remains expensive and constrains the broader usage of of the RNS strategy. E.g., the normalization procedure in [39], which may contain the right-shifting (scale down) of RNS mantissa, is intricate. A special RNS algorithm is required to support the single-digit right shifting of mantissa. To add operands with this RNS floating-point encoding, we may have to repeat the RNS right-shifting algorithm multiple times to align both mantissae. After the mantissae addition, an extra right-shifting step may be needed to normalize the sum.

The fixed-point representation is another option for defining real numbers in a computer system. The fixed-point methods typically sacrifice range and precision to trade better performance than the floating-point formats. Saokar et al. [63] utilizes the Q-format fixed-point representation with the multiplier optimization via the Urdhava Tiryakbhyam method. Essentially, they ensure the computation is the same as integer arithmetic calculation

to reduce time, space, and power overhead. However, their approach only focuses on Q15 and Q31 formats, which can only represent fractional numbers instead of real numbers. Lee, et al. [44] allows us to define fixed-point data types such as fixed_point<w,e>, where w is the width, and e is the exponent part. This fixed-point representation may process like regular integer computation. However, operand alignment may require addition or subtraction, which is considered as extra overhead compared with traditional integer computation. Yang et al. [70] extend a fixed-point type on the TVM compiler. This fixed-point type essentially is a 16-bit integer and is used to replace a 32-bit floating point to reduce the energy overhead. However, their method requires converting floating point operands to fixed-point before the convolution and finally converting the output back to floating point, which further increases the computational overhead. Besides overcoming the previously mentioned limitations from [63], [44], and [70], our *SD-Post-Mul* method (*Scaling Down Postprocessing Multiplication*) with *S-RNS-Logic-P* (*Single RNS Logical Partition; An RNS fixed-point encoding*) has more significant benefits in terms of lower latency and less energy costs. We will discuss this *SD-Post-Mul* scheme in Section 3.3. We not only ensure the integer computational benefits but also integrate the RNS strategy to further explore the delay, power, and silicon area benefits. Olsen introduced a high-precision fixed-point RNS multiplication method [53] and applied this fundamental algorithm to the neural network accelerator design. Olsen's analysis shows that his fixed-point RNS methodology obtains 7-9 times more efficient than a conventional binary matrix multiplier. The data encoding from Olsen is similar to the *D-RNS-Concat* (Double RNS Concatenation; Section 3.1), but multiplication algorithms are different. Besides the regular RNS multiplications for both integer and fractional components, Olsen's algorithm also needs a division by the fractional range, Mixed-Radix Conversions [21, 29, 36] for both integer and fractional parts, and a base-extension algorithm [33, 69] to convert the binary format back to RNS format. Compared to our *SD-Post-Mul* approach, the computing complexity is less and therefore gains better efficiency in terms of latency and energy consumption. This benefit is imperative for addition/multiplication-intensive applications with a limited energy budget, such as numerous machine learning and Edge/IoT (e.g., Federated Learning) applications.

Effective RNS fixed-point encodings can also be used as an orthogonal strategy to optimize and better support AI model quantization. The purposes of AI model quantization are to lower latency, memory storage, and energy consumption. However, significant challenges exist that limit the adoption of aggressive quantization (e.g., INT8, INT4, or INT3 [46]). Firstly, some models observed significant accuracy loss after quantification, even after applying some optimized techniques, such as Quantization Aware Training (QAT) [65]. These quantization unfriendly models include but are not limited to BERT, ResNeXt101, Mask RCNN, and GNMT [6]. Secondly, quantized weights increase the difficulty of model coverage, making training performance even worse [6]. Last but not least, there is no generalized methodology for successful quantization on most popular models, which typically requires significant human effort to fine-tune and a good understanding of the model architectures and parameters. Compared to binary adders and multipliers, RNS architectures can obtain less latency (coarse grain bit-level parallelism), power, and silicon area (fewer transistors). Therefore, RNS with effective fixed-point encoding can orthogonally further extend quantization's lower power and higher performance benefits. Similarly, for the edge/IoT devices with the latency and power budget, RNS architecture with effective fixed-point encoding can provide a better tradeoff for quantization, i.e., better data precision compared to regular binary architecture with the same resource budget. For example, the latency or power of a 4-bit (INT4) multiplier is possibly the same as that of an 8-bit RNS multiplier. This improvement allows the quantization models to achieve better precision while keeping the same resource overhead. RNS with effective fixed-point encoding essentially can provide better support for quantization to adopt more models for precision compression.

The contributions of this paper are summarized as follows:

(1) Extend implementation details of multiplication algorithms for two RNS fixed-point representations: *Double RNS Concatenation (D-RNS-Concat)* and *Single RNS Logical Partition (S-RNS-Logic-P)*. The multiplication algorithm here for the *S-RNS-Logic-P* is named as *Scaling Down Preprocessing Multiplication (SD-Pre-Mul)*.
(2) Propose an efficient *Scaling Down Postprocessing Multiplication (SD-Post-Mul)* algorithm for *S-RNS-Logic-P* fixed-point encoding, which obtains less latency and low energy while ensuring good computing accuracy.
(3) Similar to the Floating-point Unit (FPU) in regular CPUs, we design the architectures of Fixed-point Units for three multiplication algorithms: *D-RNS-Concat-Mul*, *SD-Pre-Mul*, and *SD-Post-Mul*.
(4) Set up the simulation toolchain and perform metric evaluations, demonstrating that *S-RNS-Logic-P* fixed-point encoding with *SD-Post-Mul* is a good candidate to represent real numbers for particular addition/multiplication-intensive applications (e.g., matrix multiplications in machine learning).

The remaining sections of this paper are organized as below. Section 2 gives some fundamental background of the Residue Number System (RNS). Section 3 discusses three RNS fixed-point methods, including data encodings, arithmetic algorithms, estimated overhead, and computing unit architectures. Section 4 introduces supporting algorithms and architectures that would be used in Section 3. Then in Section 5, we evaluate the efficiency of three RNS fixed-point methods. Section 6 discusses the related work, and finally, we conclude this work in Section 7.

2 BACKGROUND OF RESIDUE NUMBER SYSTEMS (RNS)

The general idea of RNS encoding is to utilize a set of smaller bit-width values to represent a larger bit-width number. When defining an RNS, we should first identify and unite n numbers m_i ($1 \le i \le n$) as a moduli set. The range of this RNS modulus set M equals the product of all moduli, where $M = \prod_{i=1}^{n} m_i$. E.g., Waston [69] selected four numbers *(199,233,194,239)* as the RNS modulus set where the range $M = 199 \times 233 \times 194 \times 239$, and $2^{31} < M < 2^{32}$. Any value X within the range $0 \le X \le M-1$ can be uniquely represented by the n nonnegative numbers (a.k.a residues). All n residues are nonnegative integers and smaller than the corresponding modulus. E.g., a decimal number *511* could be represented as $(511\%m_1, 511\%m_2, 511\%m_3, 511\%m_4) = (511\%199, 511\%233, 511\%194, 511\%239) = (113,45,123,33)$. In this RNS example, we use four 8-bit residues to represent a larger number bound by M ($M > 2^{31}$). Waston's moduli selection consists of conditions such as all moduli must be co-prime, $m_1 m_2 - m_3 m_4 = K - (K-1) = 1$, etc. The conditions of moduli selection depend on the requirements of the target system. Some efficient RNS algorithms may need additional restrictions for the modulus set, and these conditions are summarized in [69].

Each residue computes in parallel for the RNS arithmetic operations, and no carrying bit should be moved to neighbors or other residues. This isolate attribute indicates that RNS automatically obtains a certain degree of bit-level parallelism. Table 1 lists the RNS addition, subtraction, and multiplication examples with an RNS toy modulus set *(3,5,2,7)*. This RNS toy modulus set is used here for easy reading and manual computing. The RNS division requires a special algorithm which is typically considered relatively inefficient. *(1,4,1,5)* and *(1,2,1,0)* are RNS format operands, and their corresponding decimal values are *19* and *7*, respectively. For the addition, we can directly add each RNS column (residue): *(1,4,1,5) + (1,2,1,0) = ((1+1)%3, (4+2)%5, (1+1)%2, (5+0)%7) = (2,1,0,5)*. The corresponding decimal value of RNS *(2,1,0,5)* is *26* because *(26%3,26%5,26%2,26%7) = (2,1,0,5)*, which equals the decimal addition. We can also easily verify the correctness of subtraction and multiplication examples in Table 1.

Besides the inherent benefit of bit-level parallelism, RNS encoding could also simplify the architecture of integer adder and multiplier. Return to a practical RNS modulus set *(199,233,194,239)* with a range M slightly larger than 2^{31}. For a 31-bit conventional array multiplier [59], we need approximately *900 (* 30×30*)* 1-bit full adder. However, if RNS encoding is used, we will require four 8-bit array multipliers, for which the total estimated amount of 1-bit full adder is *196 (* $7 \times 7 \times 4$*)*. So besides the latency optimization, the RNS encoding can also help lower the computational units' complexity, which leads to power and area reduction. However, these RNS benefits are only directly gained from integer computations. An essential question we try to answer in this paper is how to encode real numbers in RNS while retaining the inherent latency and energy benefits.

Table 1. RNS Arithmetic Computing Examples with Toy Modulus Set (3,5,2,7); The range of this RNS $M = 3 \times 5 \times 2 \times 7 = 210$.

Op	Decimal	RNS: Residues			
		%3	%5	%2	%7
	19	1 (= 19%3)	4	1	5
	7	1 (= 7%3)	2	1	0
+	26	2 (= (1+1)%3)	1	0	5
-	12	0 (= (1-1)%3)	2	0	5
×	133	1 (= (1×1)%3)	3	1	0

Fig. 1. *Double RNS Concatenation (D-RNS-Concat)* With Toy RNS Modulus Set *(3,5,2,7)*

DEC [Int, Frac]	RNS [Int, Frac]	Represent Fixed-point Value (DEC)
[0,0]	[(0,0,0,0), (0,0,0,0)]	0.0
[0,1]	[(0,0,0,0), (1,1,1,1)]	0 + 1÷210≈ 0.0047619047619048
…	…	…
[36, 107]	[(0,1,0,1), (2,2,1,2)]	36 + 107÷210 ≈ 36.5095238095238095
…		…
[209,209]	[(2,4,1,6), (2,4,1,6)]	209 + 209÷210 ≈ 209.9952380952380952

3 RNS FIXED-POINT ENCODINGS

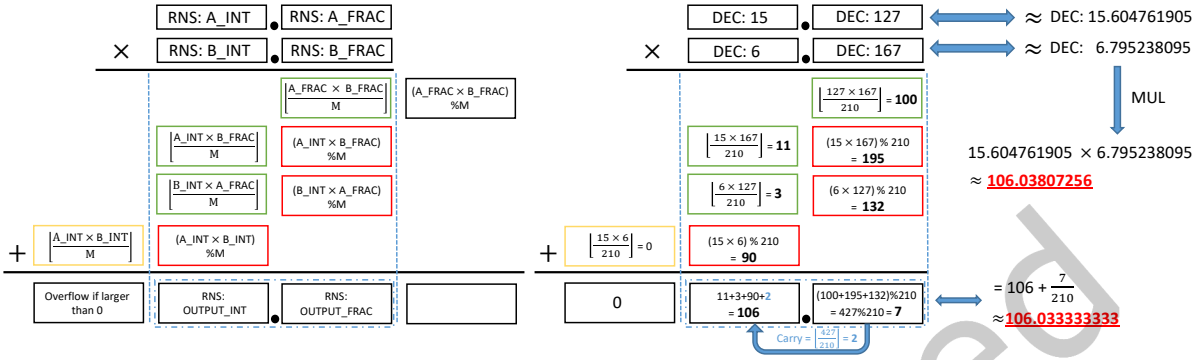
We will discuss two RNS fixed-pointed representations in this section: *Double RNS Concatenation (D-RNS-Concat)* and *Single RNS Logical Partition (S-RNS-Logic-P)*. For the *S-RNS-Logic-P* method, we can have a further classification via multiplication algorithms: *Scaling-Down Preprocessing Multiplication (SD-Pre-Mul)* and *Scaling-Down Postprocessing Multiplication (SD-Post-Mul)*. The *D-RNS-Concat* and *S-RNS-Logic-P with SD-Pre-Mul* are simply described in [23]. This section will present more details about them and further explore their attributes. Besides the algorithm details, this section also introduces each fixed-point method's overhead and multiplier architecture. To assist RNS fixed-point computation, we also require some supporting algorithms, such as *RNS Fractional Multiplication Algorithm* and *RNS Scaling-Down Algorithms*, which will be covered in Section 4.

3.1 Double RNS Concatenation (D-RNS-Concat)

Double RNS Concatenation (D-RNS-Concat) combines two RNS integers as a fixed-point real number, where the first RNS value represents the integer field while another RNS is for the fractional part. Figure 1 is the example of *D-RNS-Concat* with toy RNS modulus set *(3,5,2,7)*. Because the integer range of this modulus set $M = 3 \times 5 \times 2 \times 7 = 210$, the maximum possible value of this *D-RNS-Concat* should be less than *210 (= 210-1+1)*. And its precision is $\frac{1}{210} \approx 0.00476$. Besides the range and precision, latency and energy efficiency of arithmetic operations should also be indispensable metrics.

3.1.1 D-RNS-Concat Addition. To compute the sum, the integer and fractional RNS values of two *D-RNS-Concat* numbers could first add up in parallel. Because we concatenate two RNS values to represent a number, we must also consider the carry value from the fractional RNS sum to the integer RNS sum. This carry value has to add to the integer RNS sum as the final output. The carry value could only be either *0* or *1*, because even if adding two maximum RNS with the range, the carry value is still *1*. So, the carry value computation could be transformed to overflow detection of an integer RNS addition (fractional components). If the *D-RNS-Concat* representation is applied to Redundant Residue Number System (RRNS), the overflow of integer addition could be detected via an

Fig. 2. Double RNS Concatenation (D-RNS-Concat) Multiplication; The example of this figure utilizes toy RNS modulus set (3,5,2,7); This figure is extended from [23]

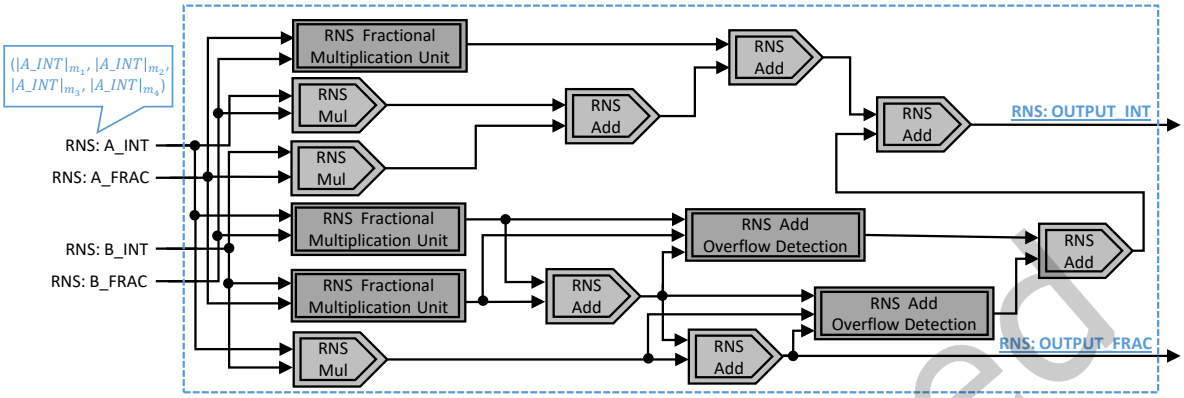


RRNS consistency checking [33, 69]. However, for the RNS without holding the redundant residues, we should utilize the Mixed-Radix Conversion methodology to compare [33, 69] and identify the addition overflow.

3.1.2 D-RNS-Concat Multiplication. The *D-RNS-Concat* multiplication and a corresponding example are presented in Figure 3. In a regular system, an arithmetic computation's output precision and range should be the same as the inputs. However, the default precision and range of the product are inconsistent with operands, which indicates extra processing is required to truncate the values. E.g., the default product of two 4-bit fixed-point binary (*2* bits for the integer and the remaining *2* bits for the fractional part) can be an 8-bit fixed-point value (*4* bits for the integer and the remaining *4* bits for the fractional part). Therefore, we should establish a regulation to automatically truncate the output to fit the same range and precision to ensure consistency. From the algorithm part (left) of Figure 3, after all the computation substeps, we have an output with four RNS values where two stand for the integer part, and the remaining two are for the fractional part. However, the input operand only employs one RNS for the integer and another RNS for the fractional number. Thus, we could retain the two RNS values close to the 'point' to construct the output and crop the remaining two RNS from both edges. Withdrawing the lowest (rightmost) RNS loses precision, but the new output precision is consistent with the input operands. Discarding the highest (leftmost) RNS may lead to erroneous results. However, the frequency of small number usage is typically much higher than large ones. For real applications, we may select an RNS modulus set with a range close to 2^{32}, which should satisfy the criteria of numerous applications. The selection rules of the RNS modulus set are discussed and summarized in [69]. Following these rules, we can also identify RNS modulus sets with ranges larger than 2^{32}. In this scenario, even if the highest RNS of the product has been truncated, the range of the integer part, which is close to 2^{32}, remains sufficient for numerous applications.

The *D-RNS-Concat Multiplication* is divided into a few subtasks. No data dependency exists among these subtasks, allowing parallel execution. In Figure 3, the green and red rectangles are the mandatory subtasks, while the yellow one is optional. The green subtasks are the *RNS Fractional Multiplication Algorithm*, which is to acquire the result of $\lfloor \frac{XY}{M} \rfloor$ and will be further introduced in Section 4.1. The red subtasks are the regular RNS integer multiplications. The optional yellow subtask also performs the *RNS Fractional Multiplication Algorithm*, but this subtask intends to detect the overflow of this whole *D-RNS-Concat Multiplication*. In the next step, we must sum up the subtask outputs in each column to construct the final product. We should also monitor the overflows during RNS integer additions to ensure that we do not lose the carry values from the fractional part to the integer part. If to further enable the overflow detection of *D-RNS-Concat Multiplication*, RNS addition overflow detections are also required when adding up the integer column.

Fig. 3. Architecture of D-RNS-Concat Multiplier; This architecture is designed based on the algorithm discussed in [23]



The right side of Figure 3 is an example of *D-RNS-Concat Multiplication* with RNS toy modulus set *(3,5,2,7)*. The range of all individual RNS integer values is [0,209]. E.g., the corresponding real number of the first *D-RNS-Concat* number *"DEC:15."DEC:127"* is *15.604761905 (DEC)*. The underline encoding of *"DEC:15"* and *"DEC:127"* in this figure essentially are two RNS numbers *(0,0,1,1)* and *(1,2,1,1)*. Considering the bijection attribute between binary/decimal integers and RNS values, we utilize decimal values in this example for easier reading. The output of the *D-RNS-Concat Multiplication* is *"DEC:106"."DEC:7"* (RNS format: *"RNS:(1,1,0,1)"."RNS:(1,2,1,0)"*), and this result stands for a fixed-point real number *106.03333333(DEC)*. Compared to the ideal product *106.03807256 (DEC)*, the difference is only *0.00473923 (DEC)*. Even though we use an RNS toy modulus set with an integer range of only *210*, *D-RNS-Concat Multiplication* can still attain good accuracy, which will be verified in Section 5.2(Figure 14) later. The absolute accuracy loss should be lower if a practical RNS modulus set is applied with a range of approximately 2^{32}. In other words, the *D-RNS-Concat* encoding attains a good value range and less accuracy loss. However, the complexity of arithmetic computing is higher than that of the *S-RNS-Logic-P* encoding, which will be introduced in Section 3.2 and Section 3.3.

3.1.3 Complexity Estimation for D-RNS-Concat. From the previous analysis, adding *2 D-RNS-Concat* numbers requires *2-3* RNS integer additions and *1* overflow detection of RNS integer addition. The *D-RNS-Concat Multiplication*, suppose the target system does not select fixed-point overflow detection, needs *3* RNS integer multiplications, *3* RNS fractional multiplications, *2* overflow detections of RNS integer addition, and *6* RNS integer additions.

3.1.4 Architecture of D-RNS-Concat Multiplier. Internal structure of *'RNS Fractional Multiplication Unit'* is illustrated in Figure 11, which we will introduce in Section 4.1. *'RNS Mul'* is the conventional RNS integer multiplier, containing a group of smaller bit-width multipliers. Unless stated otherwise, the double-line components indicate composite elements supporting specific RNS functions.

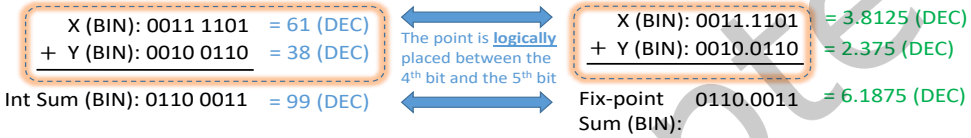
3.2 Single RNS Logical Partition(S-RNS-Logic-P): Scaling-Down Preprocessing Multiplication(SD-Pre-Mul)

3.2.1 Single RNS Logical Partition (S-RNS-Logic-P). Unlike the RNS concatenation in Section 3.1, we can merely store a fixed-point number as an RNS integer, and the system interprets this integer as a fixed-point real number by logically placing the "point" in a predetermined position. All fixed-point numbers and corresponding arithmetical outputs must follow this ordinance. E.g., an 8-bit binary *00100100* can be deciphered as *0010.0100* if the system places the logical "point" between the 4^{th} and the 5^{th} bit. We use this binary (*DEC:36*) to represent the real number *2.25*. More examples of 8-bit *Binary Logical Partition (Bin-Logic-P)* are listed in Figure 4. Within the valid RNS range, RNS numbers and binary (or decimal) numbers are injective. So same as binary, each RNS number can

Fig. 4. 8-bit *Binary Logical Partition (Bin-Logic-P)* Encoding; The logical "point" is placed between the 4^{th} and the 5^{th} bit

DEC	8-bit BIN INT	8-bit BIN Fixed-point	Represent Value (DEC)
0	00000000	0000.0000	0.0
1	00000001	0000.0001	0.0625 = $\frac{1}{2^4}$ = 1 ≫ 4
…	…	…	…
36	00100100	0010.0100	2.25 = 0.0625 × 36
…	…	…	…
99	0110 0011	0110.0011	6.1875 = 0.0625 × 99
…	…	…	…
255	11111111	1111.1111	15.9375

Fig. 5. Binary Addition of *Single RNS Logical Partition (S-RNS-Logic-P)*



also uniquely stand for a fixed-point real number. For the RNS with toy moduli set *(3,5,2,7)*, *RNS:(0,1,0,1)*, which equals *DEC:36*, could also represent the real number *2.25*. We define this data encoding as the *Single RNS Logical Partition (S-RNS-Logic-P)*. In other words, *S-RNS-Logic-P* uses only one RNS integer to describe a fixed-point real number, and this format is typically unaltered during the program execution. However, even if we only discuss this encoding with a single RNS integer, its range and precision can easily scale up by raising the number of moduli. E.g., we may reconstruct the RNS modulus set with eight 8-bit moduli to replace the one with four 8-bit moduli.

3.2.2 S-RNS-Logic-P Addition. The addition of *S-RNS-Logic-P* is less complicated than *D-RNS-Concat* because we do not require extra steps to compute the carry values from the fractional part to the integer part. *S-RNS-Logic-P* addition is equivalent to a regular RNS integer addition. Figure 5 demonstrates an example that a binary integer addition could replace a fixed-point real addition if the system follows a consistent encode/decode regulation. The encoding regulation of this example is that the logical "point" is placed between the 4^{th} bit and the 5^{th} bit of each 8-bit binary. The system interprets the integer sum *0110 0011 (BIN)* as *0110.0011 (BIN)* or *6.1875 (DEC)*, which is equivalent to the sum of the directly fixed-point addition displayed on the right side of Figure 5. This equality could also be simply verified via Figure 4. Considering the injective attribute between RNS and binary, analogous to the binary integer addition, RNS integer addition could also directly substitute this fixed-point addition. In simple words, adding two fixed-point values, which follow a predetermined logical partition regulation, can easily process like regular binary/RNS integer addition. However, the *S-RNS-Logic-P Multiplication* requires extra processing, such as scaling-down or truncating, because the raw product's range and precision are inconsistent with input operands. This section will discuss two multiplication algorithms for *S-RNS-Logic-P* encoding: *Scaling-Down Preprocessing Multiplication (SD-Pre-Mul)* and *Scaling-Down Postprocessing Multiplication (SD-Post-Mul)*.

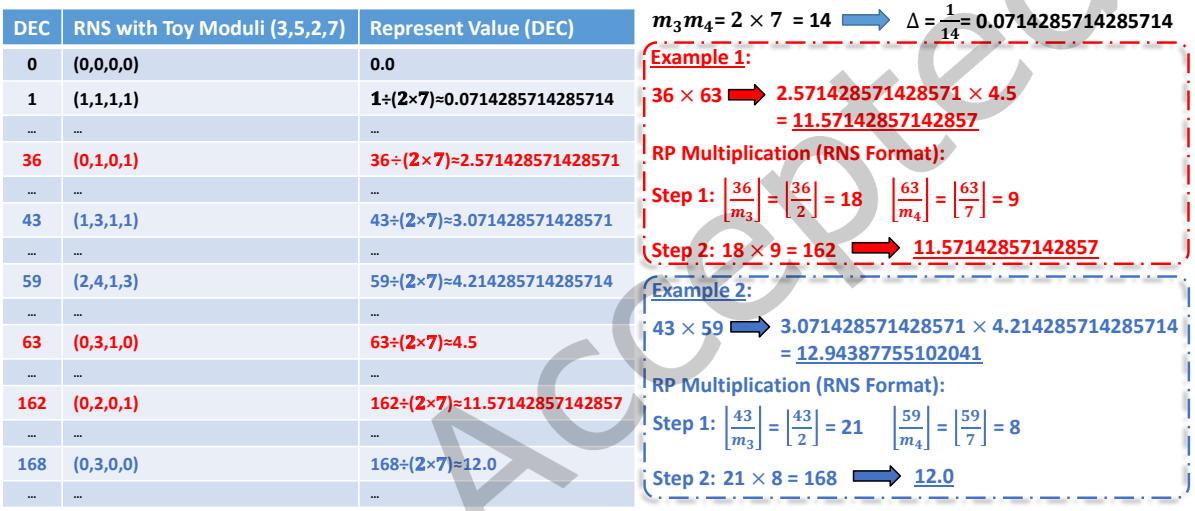
3.2.3 Scaling-Down Preprocessing Multiplication (SD-Pre-Mul). This section introduces the *SD-Pre-Mul* algorithm and further explores its limitation. The steps of the *SD-Pre-Mul* algorithm are summarized in Table 2, including a binary version for easier understanding and an RNS version.

For the binary version, the accurate product of two *n-bit* operands is at most *2n* bits. To ensure data range and precision are consistent within a system, we must crop and discard the accurate product's high $\frac{n}{2}$ bits and low $\frac{n}{2}$

Table 2. The Algorithm of *Scaling-Down Preprocessing Multiplication (SD-Pre-Mul)*

Step	Binary Version	RNS Version
1	***A*** and ***B*** are n-bit inputs; both ***A*** and ***B*** are right shifting $\frac{n}{4}$ bits (scaling-down), after which we got intermediate outputs ***A'*** and ***B'***.	***RNS_A*** and ***RNS_B*** are two RNS inputs with *4* residues each; ***RNS_A*** and ***RNS_B*** are scaling-down by m_i and m_j respectively, where $m_i m_j = K$ or $K-1$; This RNS range $M = m_1 m_2 m_3 m_4 = K(K-1)$. After scaling-down, we obtain intermediate results ***RNS_A'*** and ***RNS_B'***.
2	Compute ***A'*** × ***B'***. The product is the output of this *SD-Pre-Mul* algorithm	Compute ***RNS_A'*** × ***RNS_B'***. The product is the output of this *SD-Pre-Mul* algorithm.

Fig. 6. Examples of *SD-Pre-Mul* with RNS Toy Modulus Set *(3,5,2,7)*



bits. Assume the logical "point" is located in the middle of the *n-bit* truncated product, so omitting the low $\frac{n}{2}$ bits from the *2n-bit* accurate product may cause precision loss. For erasing the high $\frac{n}{2}$ bits from the *2n-bit* accuracy product, if the whole high $\frac{n}{2}$ bits is non-zero, then the *n-bit* truncated product should be categorized as invalid due to the overflow. After the scaling-down step (Binary Version Step 1), the low $\frac{n}{2}$ bits of the *2n-bit* raw product should have already been dropped, and no further processing is required for lower bits.

The RNS *SD-Pre-Mul* is derived from the binary version, but scaling-down factors of two operands are two distinct RNS moduli, not always $2^{\frac{n}{4}}$. From our earlier discussion, an RNS integer could bijectively represent a fixed-point real number, and we list several of these mapping examples on the left side of Figure 6 according to RNS toy modulus set *(3,5,2,7)*. On the right side of Figure 6 are two examples of *SD-Pre-Mul*. By following the RNS version algorithm in Table 2 and without loss of generality, we choose m_3 and m_4 as the scaling-down factors, where $m_3 m_4 = (K-1)$. The precision interval for this *S-RNS-Logic-P* encoding $\Delta = \frac{1}{14} \approx 0.0714$. Example 1 (red) is to calculate the product of two fixed-point real numbers, and these two operands are represented by integers *DEC:36 (RNS:(0,1,01))* and *DEC:63 (RNS:(0,3,1,0))*. In this example, we use DEC numbers to substitute RNS values for easier reading. Essentially, *S-RNS-Logic-P* process all data in RNS. By following the algorithm summarized in Table 2, for *Step 1*, the first operand *DEC:36 (RNS:(0,1,01))* is scaling-down by m_3 and the second operand *DEC:63 (RNS:(0,3,1,0))* is scaling-down by m_4. The outputs of *Step 1* are *DEC:18 (RNS:(0,3,0,4))* and *DEC:9 (RNS:(0,4,1,2))*. *Step 2* is a regular RNS integer multiplication and obtains the final output *DEC:162 (RNS:(0,2,0,1))*,

Fig. 7. Architecture of RNS *SD-Pre-Mul* Multiplier

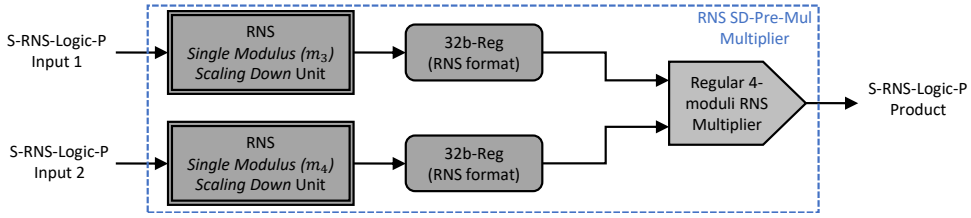


Table 3. *Scaling-Down Postprocessing Multiplication (SD-Post-Mul)*; ${O_i}^j$ denotes the j^{th} output of Step i.

Step	Binary Version	RNS Version
1	$\boldsymbol{A}$ and $\boldsymbol{B}$ are *n-bit* inputs; Compute $\lfloor \frac{AB}{M} \rfloor$ and $(AB)\%M$ in parallel; M equals 2^n, representing the value range of this system. The outputs of this step are ${\boldsymbol{O_1}}^1 = \lfloor \frac{AB}{M} \rfloor$ and ${\boldsymbol{O_1}}^2 = (AB)\%M$.	$\boldsymbol{RNS\_A}$ and $\boldsymbol{RNS\_B}$ are two RNS inputs with 4 residues each; Compute $\lfloor \frac{RNS\_A \times RNS\_B}{M} \rfloor$ and $(RNS\_A \times RNS\_B)\%M$ in parallel; M equals $m_1m_2m_3m_4$, representing the value range of this RNS. The outputs of this step are ${\boldsymbol{O_1}}^1 = \lfloor \frac{RNS\_A \times RNS\_B}{M} \rfloor$ and ${\boldsymbol{O_1}}^2 = (RNS\_A \times RNS\_B)\%M$.
2	Assuming $C = 2^{\frac{n}{2}}$; Use the output from Step 1 to make the following computations: ${\boldsymbol{O_2}}^1 = {\boldsymbol{O_1}}^1 \times C$ and ${\boldsymbol{O_2}}^2 = \frac{{\boldsymbol{O_1}}^2}{C}$.	From the conditions of RNS moduli selection, we have $K = m_1m_2$ and $K - 1 = m_3m_4$; Use the output from *Step 1* to make the following computations: ${\boldsymbol{O_2}}^1 = {\boldsymbol{O_1}}^1 \times K$ and ${\boldsymbol{O_2}}^2 = \lfloor \frac{{\boldsymbol{O_1}}^2}{K-1} \rfloor$.
3	Add the ${\boldsymbol{O_2}}^1$ and ${\boldsymbol{O_2}}^2$ to get the final output	Add the ${\boldsymbol{O_2}}^1$ and ${\boldsymbol{O_2}}^2$ to get the final output

standing for a fixed-point real number $\approx$ *11.5714*. Comparing this *S-RNS-Logic-P* output with the accurate product, we notice that both results are equal and have no accuracy loss. However, in Example 2, the difference between the *SD-Pre-Mul* output and the accurate product is significant compared with the precision interval. This portion of accuracy loss is typically intolerable for some accuracy-sensitive applications. So from Figure 6, we find that *S-RNS-Logic-P* with *SD-Pre-Mul* may lose relatively significant accuracy for certain multiplications.

3.2.4 Complexity Estimation for RNS SD-Pre-Mul. On the basis of the analysis in Section 3.2.3, the RNS *SD-Pre-Mul* algorithm only involves *2* RNS Single Modulus Scaling-Down and *1* RNS integer multiplication.

3.2.5 Architecture of RNS SD-Pre-Mul Multiplier. The architecture of the *SD-Pre-Mul* Multiplier is shown in Figure 7. The double-line rectangle *'RNS Single Modulus (m_i) Scaling-Down Unit'* is a composite unit from Figure 13, which will be described in Section 4.2.1.

3.3 Single RNS Logical Partition (S-RNS-Logic-P): Scaling-Down Postprocessing Multiplication (SD-Post-Mul)

3.3.1 Scaling-Down Postprocessing Multiplication (SD-Post-Mul). The primary defect of *SD-Pre-Mul* from Section 3.2.3 is the severe accuracy loss in certain multiplications. This issue perhaps confines the RNS to broader usage, particularly affecting applications that require better accuracy. This section introduces *Scaling-Down Postprocessing Multiplication (SD-Post-Mul)* for *S-RNS-Logic-P* encoding, with the aim of overcoming the accuracy issue while preserving decent efficiency. We name this algorithm *'postprocessing'* because the scaling-down step comes after the regular RNS integer multiplication and opposite to the *SD-Pre-Mul*, which firstly scales down both operands and followed by a regular RNS integer multiplication. The main reason for *SD-Pre-Mul*'s accuracy loss is that scaling-down operands first may lead to neglecting a subset of inputs, and the SD-Post-Mul scheme should resolve this problem.

Fig. 8. Examples of RNS *SD-Post-Mul* with RNS Toy Modulus Set *(3,5,2,7)*

DEC	RNS with Toy Moduli (3,5,2,7)	Represent Value (DEC)
0	(0,0,0,0)	0.0
1	(1,1,1,1)	1÷(2×7)≈0.0714285714285714
...	...	...
36	(0,1,0,1)	36÷(2×7)≈2.571428571428571
...	...	...
43	(1,3,1,1)	43÷(2×7)≈3.071428571428571
...	...	...
59	(2,4,1,3)	59÷(2×7)≈4.214285714285714
...	...	...
63	(0,3,1,0)	63÷(2×7)≈4.5
...	...	...
162	(0,2,0,1)	162÷(2×7)≈11.57142857142857
...	...	...
181	(1,1,1,6)	181÷(2×7)≈12.92857142857143
...	...	...

$m_3m_4 = 2 \times 7 = 14$ ⇒ $\Delta = \frac{1}{14} = 0.0714285714285714$

Example 1:

36×63 ⇒ $2.571428571428571 \times 4.5 = 11.57142857142857$

RPost Multiplication (RNS Format):

Step 1: $O_1^1 = \left\lfloor \frac{36 \times 63}{210} \right\rfloor = 10$ $O_1^2 = (36 \times 63)\%210 = 168$

Step 2: $O_2^1 = 10 \times 15 = 150$ $O_2^2 = \left\lfloor \frac{168}{14} \right\rfloor = 12$

Step 3: Result = 150 + 12 = 162 ⇒ 11.57142857142857

Example 2:

43×59 ⇒ $3.071428571428571 \times 4.214285714285714 = 12.94387755102041$

RPost Multiplication (RNS Format):

Step 1: $O_1^1 = \left\lfloor \frac{43 \times 59}{210} \right\rfloor = 12$ $O_1^2 = (43 \times 59)\%210 = 17$

Step 2: $O_2^1 = 12 \times 15 = 180$ $O_2^2 = \left\lfloor \frac{17}{14} \right\rfloor = 1$

Step 3: Result = 180 + 1 = 181 ⇒ 12.92857142857143

Fig. 9. Architecture of RNS *SD-Post-Mul* Multiplier

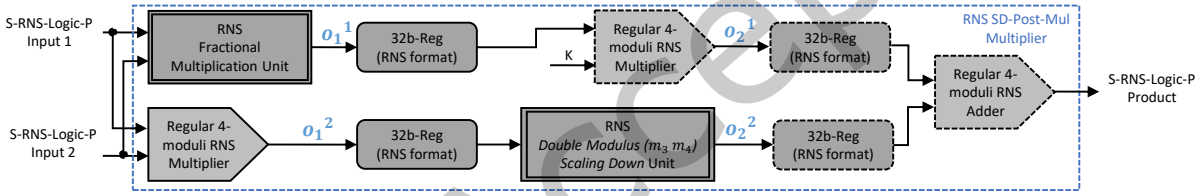
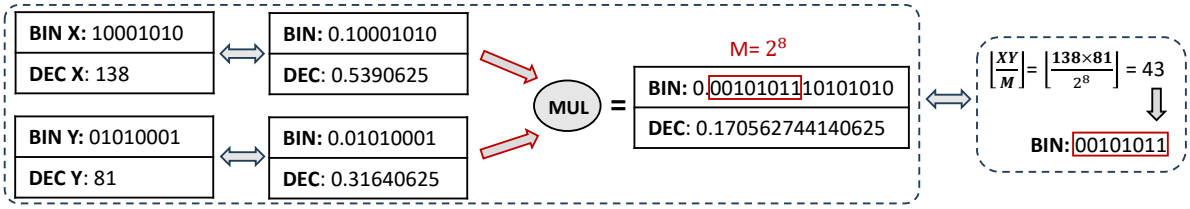


Table 3 summarizes the *SD-Post-Mul* algorithm, which could also be categorized as binary and RNS versions. Similar to *SD-Pre-Mul*, we provide the binary version to help explore the RNS version's insights more easily. However, we only utilize the RNS version for fixed-point computation. O_i^j indicates the j^{th} output of *Step i*, which is used as one of the inputs of the next step (*Step i+1*) if it exists. Figure 8 consists of two RNS *SD-Post-Mul* examples, and their inputs are equivalent to those in Figure 6 for comparison. For Example 1, the same as Figure 6, the *SD-Post-Mul* algorithm output rounds to *11.571* and matches the ideal product. For Example 2, the output of D-Post-Mul is around *12.929*, and the ideal product is *12.944*. However, when we review the *Example 1* result in Figure 6, the accuracy loss of RNS *SD-Post-Mul* drops significantly ([$\triangle_{SD-Post-Mul}$ *= (12.944 - 12.929) = 0.015*] VS [$\triangle_{SD-Pre-Mul}$ *= (12.944 - 12.0) = 0.944*]). At first glance, this granularity of accuracy loss still seems insufficient for many applications. However, in these examples, we only utilize the RNS toy modulus set *(3,5,2,7)*, which could only represent *210* integers and less than the range of an *8-bit* binary. If we switch to a practical 4-RNS modulus set (e.g., four *8-bit* moduli) with a range between 2^{31} and 2^{32}, the accuracy loss is much more applicable to real-world workloads.

3.3.2 Complexity Estimation for SD-Post-Mul. According to discussion in Section 3.3.1, the *SD-Post-Mul* for *S-RNS-Logic-P* encoding consists of *1* RNS fractional multiplication, *2* RNS integer multiplications, *1* RNS Double Modulus Scaling-Down, and *1* RNS integer addition.

3.3.3 Architecture of RNS SD-Post-Mul Multiplier. Figure 9 shows the architecture of *SD-Post-Mul* Multiplier. We will introduce *'RNS Fractional Multiplication Unit'* and *'RNS Double Moduli (m_3m_4) Scaling-Down Unit'* in Section 4.1 and Section 4.2.2, respectively.

Fig. 10. An Example of 8-bit *Binary Fractional Multiplication*



4 SUPPORTING ALGORITHMS AND ARCHITECTURES FOR RNS FIXED-POINT COMPUTATIONS

4.1 RNS Fractional Multiplication Algorithms

RNS Fractional Multiplication is a crucial subroutine of RNS fixed-point multiplication. Waston and Hasting [33, 69] proposed three relevant algorithms, one employing Mixed-Radix Conversion [21, 29, 36] and the remaining two utilizing Macrocoefficient Extraction [33, 69]. Here, we only introduce the Macrocoefficient Extraction because it requires fewer resources while preserving tolerable accuracy loss. Although Waston and Hasting gave a flow chart of Step 1, they did not explicitly discuss the remaining steps from an architectural perspective. Here, we will give more specific details about how the Macrocoefficient Extraction would be executed in an RNS fractional multiplier architecture.

4.1.1 Overview of RNS Fractional Multiplication Algorithms. *RNS Fractional Multiplication* Algorithm utilizes two RNS numbers ($|X|_{m_1}$, $|X|_{m_2}$, $|X|_{m_3}$, $|X|_{m_4}$) and ($|Y|_{m_1}$, $|Y|_{m_2}$, $|Y|_{m_3}$, $|Y|_{m_4}$) as inputs to calculate result of $\lfloor \frac{XY}{M} \rfloor$. Each RNS is interpreted as a pure fractional number ranging between *0* and *1*. Figure 10 demonstrates an example of binary edition for better reading, but essentially we employ RNS numbers instead of binary values in the underline system by following the bijective attribute. For *Input X*, the *8-bit* binary *10001010* stands for a binary fractional value *0.10001010*, and *Input Y* is similar. The accurate binary product is *0.0010101110101010*. To ensure the data range and precision consistency, we trim and preserve higher half bits after "point" *00101011* as the final product (red rectangle). This process is equivalent to calculating $\lfloor \frac{XY}{M} \rfloor$. On the right side of Figure 10, the output *BIN: 00101011* matches the left method. *RNS Fractional Multiplication* is analogous to the binary version, but the RNS value range M should be the product of all moduli.

4.1.2 RNS Fractional Multiplication via Macrocoefficient Extraction. This paper selects the Macrocoefficient Extraction approach from [33, 69], which requires less latency and hardware resources, as a subroutine of our fixed-point RNS multiplications. Requiring fewer hardware resources also implies less power consumption, which is critical for edge architecture design. As we discussed in Section 4.1.1, the inputs of *RNS Fractional Multiplication* Algorithms are two RNS integers: ($|\boldsymbol{X}|_{\boldsymbol{m_1}}$, $|\boldsymbol{X}|_{\boldsymbol{m_2}}$, $|\boldsymbol{X}|_{\boldsymbol{m_3}}$, $|\boldsymbol{X}|_{\boldsymbol{m_4}}$) and ($|\boldsymbol{Y}|_{\boldsymbol{m_1}}$, $|\boldsymbol{Y}|_{\boldsymbol{m_2}}$, $|\boldsymbol{Y}|_{\boldsymbol{m_3}}$, $|\boldsymbol{Y}|_{\boldsymbol{m_4}}$). The system interprets RNS integers as pure fractional numbers with a range between *0* and *1*. We define four macrocoefficients, $\boldsymbol{\gamma_x}$, $\boldsymbol{\delta_x}$, $\boldsymbol{\beta_y}$, and $\boldsymbol{\epsilon_y}$, as intermediate results to assist this fractional multiplication:

$$\gamma_x = \lfloor \frac{X}{m_1 m_2} \rfloor \qquad \delta_x = |X|_{m_1 m_2} \qquad \beta_y = \lfloor \frac{Y}{m_3 m_4} \rfloor \qquad \epsilon_y = |Y|_{m_3 m_4} \tag{1}$$

One prerequisite for RNS modulus set to sustain this Macrocoefficient Extraction is that $\boldsymbol{m_1 m_2 - 1 = m_3 m_4}$, implying that at least one modulus m_i must be an even number. This requirement must be incorporated when choosing the modulus set for RNS processor architecture. According to Hastings' derivation process [33], the fractional multiplication product $\lfloor \frac{XY}{M} \rfloor$ could be estimated by the following formula:

$$\lfloor \frac{XY}{M} \rfloor \approx \gamma_x \beta_y + \beta(\gamma_x \epsilon_y) + \gamma(\beta_y \delta_x) \tag{2}$$

$\gamma(input)$, $\delta(input)$, $\beta(input)$, and $\epsilon(input)$ are four general macrocoefficient calculations, where the '*input*' can be a single value or a composite equation. Essentially the definitions of γ_x, δ_x, β_y, and ϵ_y from *Formula (1)*

are special cases of general macrocoefficient calculations with *input = x* or *input = y*. According to *Formula (1)*, if the *input* = $\gamma_x\epsilon_y$, then the $\beta(input) = \beta(\gamma_x\epsilon_y) = \lfloor \frac{\gamma_x\epsilon_y}{m_3m_4} \rfloor$. Similarly, $\gamma(\beta_y\delta_x) = \lfloor \frac{\beta_y\delta_x}{m_1m_2} \rfloor$. From *Formula (2)*, the first step should calculate the value of $\boldsymbol{\gamma_x}$, $\boldsymbol{\delta_x}$, $\boldsymbol{\beta_y}$, and $\boldsymbol{\epsilon_y}$. All the above-mentioned macrocoefficients should be in RNS format throughout the *RNS Fractional Multiplication*. By following definitions in *Formula (1)* and $\boldsymbol{||X|_{m_am_b}|_{m_a} = |X|_{m_a}}$, we can easily acquire the following residue values of the macrocoefficients from the RNS inputs X and Y:

$$\begin{aligned} |\delta_x|_{m_1} &= ||X|_{m_1m_2}|_{m_1} = |X|_{m_1} & |\epsilon_y|_{m_3} &= ||Y|_{m_3m_4}|_{m_3} = |Y|_{m_3} \\ |\delta_x|_{m_2} &= ||X|_{m_1m_2}|_{m_2} = |X|_{m_2} & |\epsilon_y|_{m_4} &= ||Y|_{m_3m_4}|_{m_4} = |Y|_{m_4} \end{aligned} \tag{3}$$

According to *Formula (3)*, the values of $\boldsymbol{|\delta_x|_{m_1}}$, $\boldsymbol{|\delta_x|_{m_2}}$, $\boldsymbol{|\epsilon_y|_{m_3}}$, and $\boldsymbol{|\epsilon_y|_{m_4}}$ could be directly obtained from the input subset $\boldsymbol{|X|_{m_1}}$, $\boldsymbol{|X|_{m_2}}$, $\boldsymbol{|Y|_{m_3}}$, and $\boldsymbol{|Y|_{m_4}}$, respectively. We need extra steps to caculate the remaining residues of macrocoefficients for the inputs of *Formula (2)*: $\boldsymbol{|\delta_x|_{m_3}}$, $\boldsymbol{|\delta_x|_{m_4}}$, $\boldsymbol{|\epsilon_y|_{m_1}}$, $\boldsymbol{|\epsilon_y|_{m_2}}$, $\boldsymbol{|\gamma_x|_{m_1}}$, $\boldsymbol{|\gamma_x|_{m_2}}$, $\boldsymbol{|\gamma_x|_{m_3}}$, $\boldsymbol{|\gamma_x|_{m_4}}$, $\boldsymbol{|\beta_y|_{m_1}}$, $\boldsymbol{|\beta_y|_{m_2}}$, $\boldsymbol{|\beta_y|_{m_3}}$, and $\boldsymbol{|\beta_y|_{m_4}}$. Some intermediate results are preprocessed and stored in tiny lookup tables (LUT) to facilitate these residue computations. A terminology $\boldsymbol{ba^{\lambda}b^{(X)}}$, which is used as inputs and outputs of LUTs, is defined as below. $\boldsymbol{a}$ and $\boldsymbol{b}$ are the residue indexes.

$$ba^{\lambda}b^{(X)} = \Big||X|_{m_b} - |X|_{m_a}\Big|_{m_b} \tag{4}$$

4.1.3 $\boldsymbol{|\delta_x|_{m_3}}$ *and* $\boldsymbol{|\delta_x|_{m_4}}$ *Calculation.* To compute $\boldsymbol{|\delta_x|_{m_3}}$ and $\boldsymbol{|\delta_x|_{m_4}}$, we may firstly convert their representations via the following deduction:

$$\begin{aligned} |\delta_x|_{m_3} &= \Bigg||\delta_x|_{m_1} + \Big||\delta_x|_{m_3} - |\delta_x|_{m_1}\Big|_{m_3}\Bigg|_{m_3} = \Big||\delta_x|_{m_1} + 31^{\lambda}3^{(\delta_x)}\Big|_{m_3} \\ |\delta_x|_{m_4} &= \Bigg||\delta_x|_{m_1} + \Big||\delta_x|_{m_4} - |\delta_x|_{m_1}\Big|_{m_4}\Bigg|_{m_4} = \Big||\delta_x|_{m_1} + 41^{\lambda}4^{(\delta_x)}\Big|_{m_4} \end{aligned} \tag{5}$$

From *Formula (1)*, $\boldsymbol{|\delta_x|_{m_1}}$ equals to the $\boldsymbol{|X|_{m_1}}$, which is a subset of algorithm inputs. To raise algorithm efficiency, a lookup table (LUT) is utilized to provide the values of $\boldsymbol{31^{\lambda}3^{(\delta_x)}}$ and $\boldsymbol{41^{\lambda}4^{(\delta_x)}}$. The LUT input is $\boldsymbol{21^{\lambda}2^{(\delta_x)}}$ that maintains as an index column of the table. The row count of this LUT is equal to $\boldsymbol{m_2}$. If the underline system employs the RNS toy modulus set *(3,5,2,7)*, we have $\boldsymbol{m_2} = 5$, indicating the row count of this LUT is *5*. Table 4 shows the *m_2−LUT* with RNS toy modulus set *(3,5,2,7)*. The input of *m_2−LUT*, $\boldsymbol{21^{\lambda}2^{(\delta_x)}}$, we can calculate via *Formula (3)* and *Formula (4)*:

$$21^{\lambda}2^{(\delta_x)} = \Big||(\delta_x)|_{m_2} - |(\delta_x)|_{m_1}\Big|_{m_2} = \Big||X|_{m_2} - |X|_{m_1}\Big|_{m_2} \tag{6}$$

$\boldsymbol{21^{\lambda}2^{(\delta_x)}}$	$\boldsymbol{31^{\lambda}3^{(\delta_x)}}$	$\boldsymbol{41^{\lambda}4^{(\delta_x)}}$
0	0	0
1	0	6
2	0	5
3	1	3
4	1	2

Table 4. *m_2−LUT* with RNS Modulus Set *(3,5,2,7)*; $m_2 = 5$

$\boldsymbol{43^{\lambda}4^{(\gamma_x)}}$	$\boldsymbol{13^{\lambda}1^{(\gamma_x)}}$	$\boldsymbol{23^{\lambda}2^{(\gamma_x)}}$
0	0	0
1	2	3
2	2	2
3	1	0
4	1	4
5	0	2
6	0	1

Table 5. *m_4−LUT* with RNS Toy Modulus Set *(3,5,2,7)*; $m_4 = 7$

So the input of m_2−*LUT*, $\mathbf{21^{\lambda}2^{(\delta_x)}}$,could be computed via the *Formula (6)* and *RNS Fractional Multiplication* inputs ($|X|_{m_1}$ and $|X|_{m_2}$).

4.1.4 $\boldsymbol{|\epsilon_y|_{m_1}}$ *and* $\boldsymbol{|\epsilon_y|_{m_2}}$ *Calculation.* The computation of $\boldsymbol{|\epsilon_y|_{m_1}}$ and $\boldsymbol{|\epsilon_y|_{m_2}}$ is similar to $\boldsymbol{|\delta_x|_{m_3}}$ and $\boldsymbol{|\delta_x|_{m_4}}$:

$$|\epsilon_y|_{m_1} = \left||\epsilon_y|_{m_3} + \left||\epsilon_y|_{m_1} - |\epsilon_y|_{m_3}\right|_{m_1}\right|_{m_1} = \left||\epsilon_y|_{m_3} + 13^{\lambda}1^{(\epsilon_y)}\right|_{m_1}$$
$$|\epsilon_y|_{m_2} = \left||\epsilon_y|_{m_3} + \left||\epsilon_y|_{m_2} - |\epsilon_y|_{m_3}\right|_{m_2}\right|_{m_2} = \left||\epsilon_y|_{m_3} + 23^{\lambda}2^{(\epsilon_y)}\right|_{m_2} \tag{7}$$

From *Formula (3)*, $\boldsymbol{|\epsilon_y|_{m_3}}$ and $\boldsymbol{|\epsilon_y|_{m_4}}$ are equal to $|Y|_{m_3}$ and $|Y|_{m_4}$, respectively, which are a subset of inputs for *RNS Fractional Multiplication* Algorithm. The results of $\mathbf{13^{\lambda}1^{(\epsilon_y)}}$ and $\mathbf{23^{\lambda}2^{(\epsilon_y)}}$ in *Formula (7)* could be obtained by accessing a similar m_4−*LUT*(E.g., Table 5 with RNS Modulus Set *(3,5,2,7)*). The value range of $\boldsymbol{\gamma_x}$ and $\boldsymbol{\epsilon_y}$ are both [0, m_3m_4-1]. Thus, we can make use of $\mathbf{43^{\lambda}4^{(\epsilon_y)}}$ as the input to access m_4-*LUT* for $\mathbf{13^{\lambda}1^{(\epsilon_y)}}$ and $\mathbf{23^{\lambda}2^{(\epsilon_y)}}$. Similar to *Formula (6)*, $\mathbf{43^{\lambda}4^{(\epsilon_y)}}$ is calculated via the following equation:

$$43^{\lambda}4^{(\epsilon_y)} = \left||(\epsilon_y)|_{m_4} - |(\epsilon_y)|_{m_3}\right|_{m_4} = \left||Y|_{m_4} - |Y|_{m_3}\right|_{m_4} \tag{8}$$

4.1.5 $\boldsymbol{|\gamma_x|_{m_1}}$, $\boldsymbol{|\gamma_x|_{m_2}}$, $\boldsymbol{|\gamma_x|_{m_3}}$ *and* $\boldsymbol{|\gamma_x|_{m_4}}$ *Calculation.* According to the proof from Hastings [33], $\boldsymbol{|\gamma_x|_{m_3}}$ and $\boldsymbol{|\gamma_x|_{m_4}}$ could transform to the following formats:

$$|\gamma_x|_{m_3} = \left|31^{\lambda}3^{(X)} - 31^{\lambda}3^{(\delta_x)}\right|_{m_3} \qquad |\gamma_x|_{m_4} = \left|41^{\lambda}4^{(X)} - 41^{\lambda}4^{(\delta_x)}\right|_{m_4} \tag{9}$$

Analogous to the $\boldsymbol{|\delta_x|_{m_3}}$ and $\boldsymbol{|\delta_x|_{m_4}}$ computation, the values of $\mathbf{31^{\lambda}3^{(\delta_x)}}$ and $\mathbf{41^{\lambda}4^{(\delta_x)}}$ could be fetched from the m_2-*LUT*. The value of $\mathbf{31^{\lambda}3^{(X)}}$ and $\mathbf{41^{\lambda}4^{(X)}}$ could simply calculate by using $\boldsymbol{|X|_{m_1}}$, $\boldsymbol{|X|_{m_3}}$, and $\boldsymbol{|X|_{m_4}}$ via *Formula (4)*. The computations of $\boldsymbol{|\gamma_x|_{m_1}}$ and $\boldsymbol{|\gamma_x|_{m_2}}$ are to use the result of $\boldsymbol{|\gamma_x|_{m_3}}$:

$$|\gamma_x|_{m_1} = \left||\gamma_x|_{m_3} + \left||\gamma_x|_{m_1} - |\gamma_x|_{m_3}\right|_{m_1}\right|_{m_1} = \left||\gamma_x|_{m_3} + 13^{\lambda}1^{(\gamma_x)}\right|_{m_1}$$
$$|\gamma_x|_{m_2} = \left||\gamma_x|_{m_3} + \left||\gamma_x|_{m_2} - |\gamma_x|_{m_3}\right|_{m_2}\right|_{m_2} = \left||\gamma_x|_{m_3} + 23^{\lambda}2^{(\gamma_x)}\right|_{m_2} \tag{10}$$

The $\mathbf{13^{\lambda}1^{(\gamma_x)}}$ and $\mathbf{23^{\lambda}2^{(\gamma_x)}}$ in *Formula (10)* are collected from m_4-*LUT*, which employs $\mathbf{43^{\lambda}4^{(\gamma_x)}}$ as this LUT index. E.g., Table 5 shows a *m4-LUT* with RNS toy modulus set *(3,5,3,7)*. The index $\mathbf{43^{\lambda}4^{(\gamma_x)}}$ could also be easily computed by using $\boldsymbol{|\gamma_x|_{m_3}}$ and $\boldsymbol{|\gamma_x|_{m_4}}$ by following *Formula (4)*.

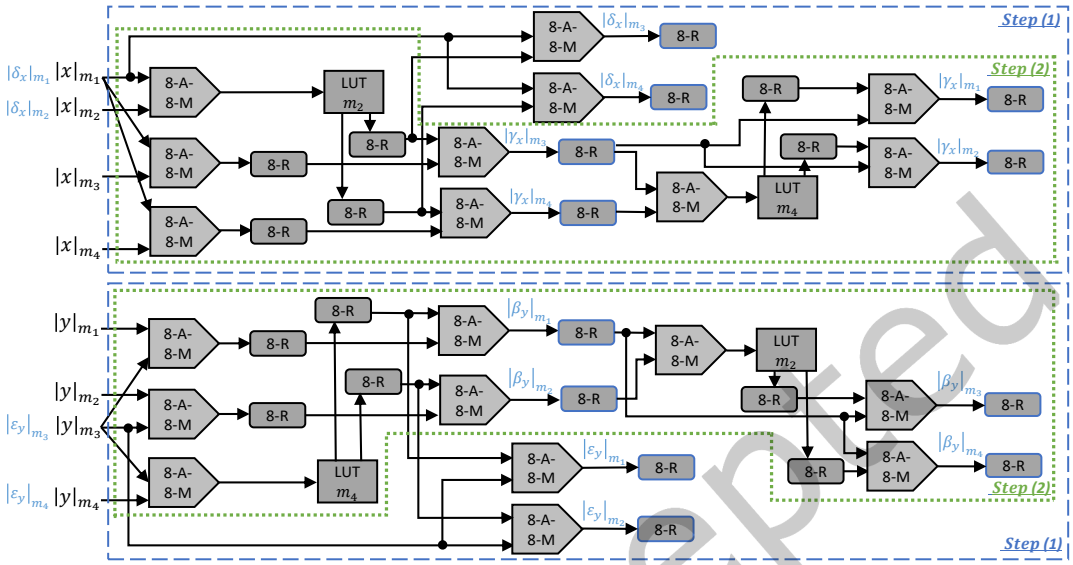
4.1.6 $\boldsymbol{|\beta_y|_{m_1}}$, $\boldsymbol{|\beta_y|_{m_2}}$, $\boldsymbol{|\beta_y|_{m_3}}$, *and* $\boldsymbol{|\beta_y|_{m_4}}$ *Calculation.* The computation of $\boldsymbol{|\beta_y|_{m_1}}$, $\boldsymbol{|\beta_y|_{m_2}}$, $\boldsymbol{|\beta_y|_{m_3}}$, and $\boldsymbol{|\beta_y|_{m_4}}$ are similar to that of $\boldsymbol{|\gamma_x|_{m_1}}$, $\boldsymbol{|\gamma_x|_{m_2}}$, $\boldsymbol{|\gamma_x|_{m_3}}$ and $\boldsymbol{|\gamma_x|_{m_4}}$ in Section 4.1.5. $\boldsymbol{|\beta_y|_{m_1}}$ and $\boldsymbol{|\beta_y|_{m_2}}$ are represented as follow:

$$|\beta_y|_{m_1} = \left|13^{\lambda}1^{(\epsilon_y)} - 13^{\lambda}1^{(Y)}\right|_{m_1} \qquad |\beta_y|_{m_2} = \left|23^{\lambda}2^{(\epsilon_y)} - 23^{\lambda}2^{(Y)}\right|_{m_2} \tag{11}$$

To compute $\boldsymbol{|\beta_y|_{m_3}}$ and $\boldsymbol{|\beta_y|_{m_4}}$, we use the $\boldsymbol{|\beta_y|_{m_1}}$ from *Formula (11)* as a subset of inputs:

$$|\beta_y|_{m_3} = \left||\beta_y|_{m_1} + \left||\beta_y|_{m_3} - |\beta_y|_{m_1}\right|_{m_3}\right|_{m_3} = \left||\beta_y|_{m_1} + 31^{\lambda}3^{(\beta_y)}\right|_{m_3}$$

Fig. 11. The Architecture of *RNS Fractional Multiplier* with four moduli. All moduli (m_i) in this diagram are *8-bit*; *8-A-8-M* represents a logic of an *8-bit* adder followed by an *8-bit* modulo; *8-R* stands for an *8-bit* register. *LUT* m_i is a lookup table with m_i entries.



$$|\beta_y|_{m_4} = \left| |\beta_y|_{m_1} + \left| |\beta_y|_{m_4} - |\beta_y|_{m_1} \right|_{m_4} \right|_{m_4} = \left| |\beta_y|_{m_1} + 41^{\lambda} 4^{(\beta_y)} \right|_{m_4} \tag{12}$$

The remaining terms of *Formula (12)*, $\mathbf{31^{\lambda}3^{(\beta_y)}}$ and $\mathbf{41^{\lambda}4^{(\beta_y)}}$, could be read from the same m_2-*LUT* via an index $\mathbf{21^{\lambda}2^{(\beta_y)}}$. The m_2-*LUT* is possible to be shared because the value range of $\boldsymbol{\delta_x}$ and $\boldsymbol{\beta_y}$ is equivalent ([0, m_1m_2-1]).

4.1.7 Remaining Steps Of RNS Fractional Multiplication via Macrocoefficient Extraction. From Section 4.1.2 to Section 4.1.6, we introduced the residue computation of all the macrocoefficients. According to *Formula (2)*, in the following steps, we will require to calculate β_A and γ_B, where $A = \gamma_x\epsilon_y$ and $B = \beta_y\delta_x$. Because A and B are also in RNS format, we can utilize the same method to determine β_A and γ_B. Finally, we add up all the products as the final result of $\lfloor \frac{XY}{M} \rfloor$.

4.1.8 Architecture of RNS Fractional Multiplier. We introduce *RNS Fractional Multiplication* Algorithm (Output: $\lfloor \frac{XY}{M} \rfloor$) in Section 4.1. Many modern processors integrated one or more Floating-Point Units (FPUs) [20, 42, 47, 55, 72] to boost the performance and efficiency of floating-point operations. Thus the architecture of *RNS Fractional Multiplication* unit is crucial to realize high-performance and low-cost RNS fixed-point computation. Hastings [33] provide flowcharts to compute $\boldsymbol{\gamma_x}$, $\boldsymbol{\delta_x}$, $\boldsymbol{\beta_y}$, and $\boldsymbol{\epsilon_y}$. Based on Hasting's flowchart, we provide the architecture design of *RNS Fractional Multiplication Unit* in Figure 11. Moreover, we explicitly identify which subcomponents of this architecture could be applied to calculate this algorithm's following step (*Step 2*) after the macrocoeffficent computation (*Step 1*). The components within the blue rectangle are for macrocoeffficent computation (*Step 1*). The *Step 1* outputs (macrocoeffficent values) are explicitly marked in Figure 11. *Step 2* is to compute β and γ. Thus a subset of architectural components is reusable for different steps' computation. After *Step 2*, according to *Formula (2)*, we add up three products to get the final *RNS Fractional Multiplication* result: $\lfloor \frac{XY}{M} \rfloor$.

Table 6. Three-input *RNS Base-extension* Algorithm From Waston and Hasting [33, 69]

Step	Description
0	The three inputs are $\lvert\frac{X}{m_4}\rvert_{m_1}$, $\lvert\frac{X}{m_4}\rvert_{m_2}$, and $\lvert\frac{X}{m_4}\rvert_{m_3}$
1	Compute $21^{\lambda}2^{(\delta\frac{X}{m_4})} = \left\lvert\left\lvert\delta\frac{X}{m_4}\right\rvert_{m_2} - \left\lvert\delta\frac{X}{m_4}\right\rvert_{m_1}\right\rvert_{m_2} = \left\lvert\lvert\frac{X}{m_4}\rvert_{m_2} - \lvert\frac{X}{m_4}\rvert_{m_1}\right\rvert_{m_2}$
2	Use $21^{\lambda}2^{(\delta\frac{X}{m_4})}$ as an index to access m_2-*LUT* (E.g., Table 4 with RNS toy modulus set) to fetch the values of $31^{\lambda}3^{(\delta\frac{X}{m_4})}$ and $41^{\lambda}4^{(\delta\frac{X}{m_4})}$
3	Compute $\left\lvert\delta\frac{X}{m_4}\right\rvert_{m_3}$ and $\left\lvert\delta\frac{X}{m_4}\right\rvert_{m_4}$ via equations $\left\lvert\delta\frac{X}{m_4}\right\rvert_{m_3} = \left\lvert 31^{\lambda}3^{(\delta\frac{X}{m_4})} + \left\lvert\delta\frac{X}{m_4}\right\rvert_{m_1}\right\rvert_{m_3}$ and $\left\lvert\delta\frac{X}{m_4}\right\rvert_{m_4} = \left\lvert 41^{\lambda}4^{(\delta\frac{X}{m_4})} + \left\lvert\delta\frac{X}{m_4}\right\rvert_{m_1}\right\rvert_{m_4}$, where $\left\lvert\delta\frac{X}{m_4}\right\rvert_{m_1} = \left\lvert\frac{X}{m_4}\right\rvert_{m_1}$
4	Compute $\gamma_{\frac{X}{m_4}} = \left\lvert\lvert\frac{1}{m_1m_2}\rvert_{m_3} \times \lvert\frac{X}{m_4}\rvert_{m_3} - \left\lvert\delta\frac{X}{m_4}\right\rvert_{m_3}\right\rvert_{m_3}$
5	Compute $\left\lvert\frac{X}{m_4}\right\rvert_{m_4} = \left\lvert m_1m_2 \times \gamma_{\frac{X}{m_4}}\right\rvert_{m_4} + \left\lvert\delta\frac{X}{m_4}\right\rvert_{m_4}\Big\rvert_{m_4}$

Fig. 12. RNS Scaling-Down (By A Single Modulus) Algorithm With RNS Toy Modulus Set *(3,5,2,7)*

		DEC	RNS Residues X%3	X%5	X%2	X%7	Formula (1) Computation			Three Inputs Based Extension
Example 1	X	15	0	0	1	1				
	$\lvert\frac{X}{m_4}\rvert$	2	2	2	0	2	$\lvert 1 * \lvert 0-1\rvert_3\rvert_3 = 2$	$\lvert 3 * \lvert 0-1\rvert_5\rvert_5 = 2$	$\lvert 1 * \lvert 1-1\rvert_2\rvert_2 = 0$	(1) $\lvert\frac{1}{m_4}\rvert_{m_4}$, No solution (2) $\lvert\frac{X}{m_4}\rvert_{m_4} = \lvert 0 + \lvert\delta\frac{X}{m_4}\rvert_{m_4}\rvert_{m_4} = 2$
Example 2	X	67	1	2	1	4				
	$\lvert\frac{X}{m_4}\rvert$	9	0	4	1	2	$\lvert 1 * \lvert 1-4\rvert_3\rvert_3 = 0$	$\lvert 3 * \lvert 2-4\rvert_5\rvert_5 = 4$	$\lvert 1 * \lvert 1-4\rvert_2\rvert_2 = 1$	(1) $\lvert\frac{1}{m_4}\rvert_{m_4}$, No solution (2) $\lvert\frac{X}{m_4}\rvert_{m_4} = \lvert 0 + \lvert\delta\frac{X}{m_4}\rvert_{m_4}\rvert_{m_4} = 2$

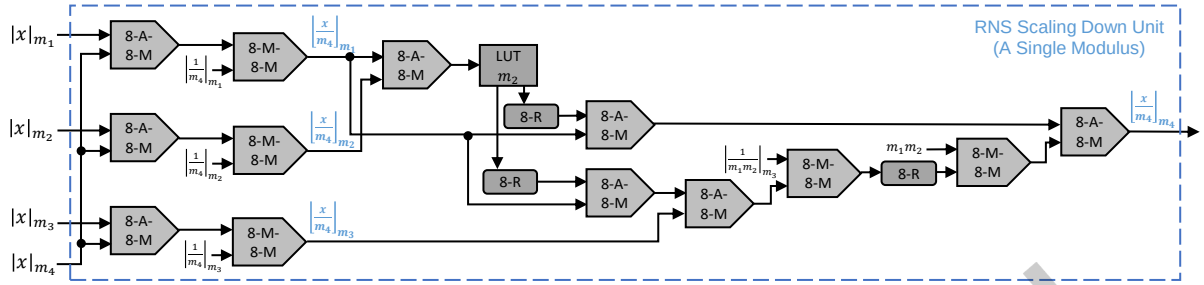
4.2 RNS Scaling-Down By A Single Modulus Or By The Product Of Two Moduli

The RNS scaling-down and division are generally categorized as heavy cost operations. However, in some specific scenarios, the overhead of scaling-down particular constants may be tolerable. These special constants include a single RNS modulus and the product of the RNS modulus subset. The scaling-down by these constants should be less complicated than by 2^n. Waston and Hasting [33, 69] proposed an RNS scaling-down algorithm by a factor of single or multiple moduli (product). This section introduces Waston and Hasting's RNS scaling-down algorithm, which is a subtask of RNS *SD-Pre-Mul* or *SD-Post-Mul* for *S-RNS-Logic-P* encoding.

4.2.1 RNS Scaling-Down By A Single Modulus. From Section 3.2, the *SD-Pre-Mul* requires a subroutine to scale down an RNS value by a single modulus. Without loss of generality, in an RNS system with four moduli, we assume that the single modulus we scale down is m_4. The input of this scaling-down algorithm is ($\lvert X\rvert_{m_1}$, $\lvert X\rvert_{m_2}$, $\lvert X\rvert_{m_3}$, $\lvert X\rvert_{m_4}$), and the output should be ($\lvert\frac{X}{m_4}\rvert_{m_1}$, $\lvert\frac{X}{m_4}\rvert_{m_2}$, $\lvert\frac{X}{m_4}\rvert_{m_3}$, $\lvert\frac{X}{m_4}\rvert_{m_4}$). From the proof of Waston and Hasting [33, 69], if the m_i does not share common factors with m_4, then $\lvert\frac{X}{m_4}\rvert_{m_i}$ could be simply obtained via the following equation:

$$\lfloor\frac{X}{m_4}\rfloor_{m_i} = \left\lvert\lvert\frac{1}{m_4}\rvert_{m_i} * \left\lvert\lvert X\rvert_{m_i} - \lvert X\rvert_{m_4}\right\rvert_{m_i}\right\rvert_{m_i} \tag{13}$$

Fig. 13. Architecture of RNS *Single Modulus Scaling-Down* Unit



We can utilize a three-input *base-extension* algorithm [33, 69] for the remaining residue(s) that fail to satisfy the above criteria. Table 6 summarizes the *base-extension* algorithm steps for RNS with four moduli.

Figure 12 includes two examples of the single modulus scaling-down algorithm with RNS toy modulus set *(3,5,2,7)*. Assuming the single modulus to scale down is m_4, and the output is the RNS format of $\lfloor \frac{X}{m_4} \rfloor$.

Example 1: For the regular decimal format, if the input $X = 15$, then we have $\lfloor \frac{X}{m_4} \rfloor = 2$. The equivalent RNS values of the previous two decimal numbers are *(0,0,1,1)* and *(2,2,0,2)*, respectively. The above RNS scaling-down algorithm helps us to obtain the output *(2,2,0,2)* from input *(0,0,1,1)*. We can directly employ *Formula (13)* to compute the first three residues. For $i = 1, 2, 3$, $\lfloor \frac{X}{m_4} \rfloor_{m_i}$ equals *2,2* and *0*, respectively, which match the corrected residues. However, we cannot make use of *Formula (13)* to calculate the fourth residue because there is no solution for $\left|\frac{1}{m_4}\right|_{m_4}$. To obtain the fourth residue, we need to use the *base-extension* algorithm, where the three algorithm inputs are the outputs of *Formula (13)*. In this example, *2,2,* and *0* are inputs, and the *base-extension* algorithm output is *2*, consistent with the corrected result.

Example 2: Similar to *Example 1*, we switch the decimal $X = 67$, and then $\lfloor \frac{X}{m_4} \rfloor = 9$. After the apply the same methodology as *Example 1*, we can obtain the correct RNS scaling-down result *(0,4,1,2)*.

4.2.2 RNS Scaling-Down By The Product Of Two Moduli. *Step 2* of *SD-Post-Mul* (Section 3.3) requires to scale down by *K - 1* (= m_3m_4). Scaling down by the product of two moduli, which equals to *K* or *K-1*, should be less complicated because it associates with the definition of macrocoefficients (Section 4.1.2): $\gamma_x = \lfloor \frac{X}{m_1m_2} \rfloor$ and $\beta_x = \lfloor \frac{X}{m_3m_4} \rfloor$. If the scaling-down factor is *K - 1* (= m_3m_4), we only need to compute the RNS version of β_x. To obtain the residues of β_x, we can simply employ a subset of the architecture in Figure 11 , i.e., to include the datapath for $|\beta_y|_{m_1}$, $|\beta_y|_{m_2}$, $|\beta_y|_{m_3}$, and $|\beta_y|_{m_4}$.

4.2.3 Architecture of RNS Single Modulus Scaling-Down Unit. The RNS *SD-Pre-Mul* algorithm incorporates *Single Modulus Scaling-Down* as subtasks, and the associated architecture of this logic is shown in Figure 13. The RNS *SD-Post-Mul* algorithm needs to scale down the product of two moduli. More specifically, it needs to scale down by *K - 1* or *K*, which makes this scaling-down operation associated with macrocoefficients. According to the previous discussion, the architecture of this scaling-down unit (by *K - 1*, a specific case of the product of two moduli) should incorporate the logic and datapath in Figure 11 that calculate the results of $|\beta_y|_{m_1}$, $|\beta_y|_{m_2}$, $|\beta_y|_{m_3}$, and $|\beta_y|_{m_4}$. In other words, the architecture of the *Scaling-Down Unit (by K - 1)* is only a subset of the *RNS Fractional Multiplier* architecture.

5 EVALUATION

5.1 Evaluation Methodology

This section measures RNS fixed-point approaches from the following four metrics: computational accuracy, latency, energy, and Energy-Delay Product (EDP). In Section 5.2, we evaluate the accuracy of three multiplication algorithms with RNS toy and practical modulus sets. Because the data range of the RNS toy modulus set is small, we can use brute force to evaluate all possible two-number pairs. Each two-number pair includes the two

operands of every arithmetic operation. However, the RNS practical modulus set has a much larger valid range, so we randomly pick 4096 two-number pairs for accuracy assessments. For the remaining metrics, i.e., latency, energy, and Energy-Delay Product (EDP) in Section 5.3, 5.4 and 5.5, we use gem5 [13] as a front-end of simulation toolchain to generate the ARM assembly traces. The binary executable file of each workload is cross-complied and collected as one of the gem5 inputs. Gem5's debug mode output contains assembly instructions of the specific workload. Then these assembly traces are filled into a traced-based in-order RNS CPU simulator to measure the latency and energy of each workload. The basic events and parameter values of the RNS CPU simulator are collected from gem5. By integrating these events and parameters, our RNS CPU simulator provides better encapsulation and more flexibility to model the RNS components and RNS processors. The accuracies of gem5 have been systemically verified in related work [32] and [18], which meet the simulation requirements of computer architecture design for academia and industry. To ensure the measurement accuracies of the RNS CPU simulator, besides utilizing the basic event and parameter values from gem5, we also integrate correction suggestions from [8] to further enhance the simulation accuracies. Therefore, the simulation accuracies of the RNS CPU simulator should also meet the measurement requirements for both academics and industry. The evaluation workloads of this work incorporate 5 benchmarks from nanobench [4], and the remaining 6 test cases are convolutional computations of machine learning models. These machine learning models include alexnet [43], FaceRecognition [2], mobilenet [35], OCR [2], resnet [34] and yolo_tiny [68]. Considering the similarity of the matrix multiplication patterns in layers and reducing the simulation timing, we pick the first convolutional layer of each machine learning model to represent the associated model.

5.2 Accuracy Measurement

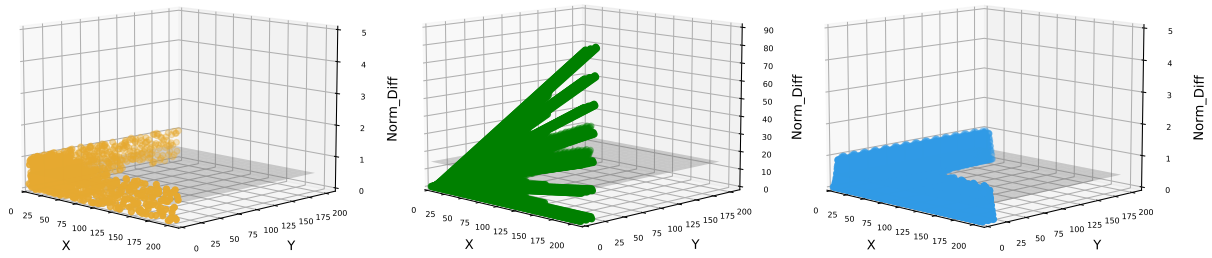
Computational accuracy is one of the critical evaluation metrics for data encoding. Unlike the additions, the raw multiplication outputs of three fixed-point encodings are problematic because the raw products' ranges and precisions are always inconsistent with operands. So we must cautiously discard a subset of data within the multiplication process while aiming to minimize the accuracy loss. The accuracy loss is the absolute value of the difference between the correct result and the algorithm output. If applying RNS fixed-point encoding to AI applications, we must use the strategy with limited accuracy loss. AI models typically include multiple layers, and each layer contains a huge number of arithmetic multiplications. If the accuracy loss of a single multiplication is non-negligible, the error will accumulate layer by layer, making the final result unacceptable. So, the algorithm with large accuracy loss in a single multiplication, such as *SD_Pre_Mul*, is unsuitable for AI applications. The accumulated errors will make AI models challenging to converge in training and significant accuracy degradation in inference.

Figure 14 compares the accuracy of three multiplications with the RNS toy modulus set *(3,5,2,7)*. Multiplication overflow indicates an incorrect product and is unnecessary to incorporate into statistics. Because the range of the RNS toy modulus set is small, Figure 14 includes all valid products (no overflow) of two-number pairs. The definition of *Norm_Diff* in Figure 14 is as below:

Definition 1 (Normalized difference): *Normalized difference (Norm-diff)* is defined by the following formula, in which the *Min_Enc_Interval* is the minimal precision interval that this data encoding could represent:

$$\boldsymbol{Norm\_Diff} = \frac{\left|(Ideal\_Product - Multiplication\_Output)\right|}{(Min\_Enc\_Interval)}$$

According to the above definition, a smaller *Norm_Diff* implies better computational accuracy. Figure 15 demonstrates the accuracy (*Norm_Diff*) of three multiplications with practical RNS modulus set *(199,233,194,239)* [69]. The range of this practical modulus set is close to 2^{31} and can meet the range and precision requirements of many IoT or embedded system applications. Unlike Figure 14, which tests all two-number pairs by brute force, Figure 15 randomly selects *4096* pairs to represent the product accuracy due to the enormous exploration space of this practical RNS modulus set. Similarly, Figure 15 does not contain the invalid overflow products. Because

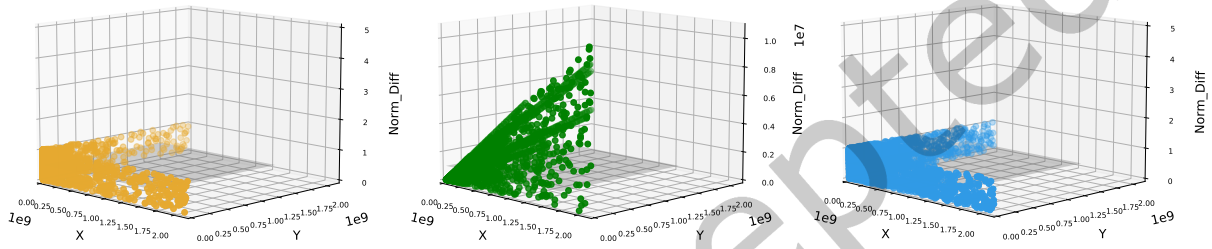


(a) D_RNS_Concat_Mul
Mean Norm_Diff: 0.480

(b) SD_Pre_Mul
Mean Norm_Diff: 14.212

(c) SD_Post_Mul
Mean Norm_Diff: 0.392

Fig. 14. Accuracy *Norm_Diff* For RNS *(3,5,2,7)*



(a) D_RNS_Concat_Mul
Mean Norm_Diff: 0.494

(b) SD_Pre_Mul
Mean Norm_Diff: 992918.047

(c) SD_Post_Mul
Mean Norm_Diff: 0.497

Fig. 15. Accuracy *Norm_Diff* For RNS *(199,233,194,239)*

a *D_RNS_Concat* number consists of an integer RNS and a fractional RNS, to simplify the evaluation space in Figure 14, only the integer components utilize the brute force selection while the fractional values are randomly chosen. The data patterns of the toy and practical modulus sets are similar. The mean *Norm_Diff* values of both *D_RNS_Concat_Mul* and *SD_Post_Mul* are very close to *0.5*, which implies these two multiplication strategies are excellent for keeping the product accuracy. However, for the practical RNS moduli set *(199,233,194,239)*, the mean *Norm_Diff* of *SD_Pre_Mul* is *992918.047*, significantly higher than the other two methods. For the correctness of the IEEE standard(IEEE 754), if the operand pairs are uniform distribution, the mean *Norm_Diff* of IEEE 754 should be very close to $0.5 \times Precision\_Interval$, the same as the mean *Norm_Diff* of *D_RNS_Concat_Mul* and *SD_Post_Mul*. This result indicates that *SD_Pre_Mul* may lose more product information and has much worse accuracy than *D_RNS_Concat_Mul* and *SD_Post_Mul*.

5.3 Latency Measurement

The latency to complete a practical workload is a metric to evaluate the performance of data encodings. Figure 16 presents the normalized latencies of the three RNS fixed-point schemes. To ensure the latency comparison is more straightforward, the latencies of both *S-RNS-Logic-P* methods (*SD-Pre-Mul* and *SD-Post-Mul*) are normalized to that of *D-RNS-Concat*. From the results, the *S-RNS-Logic-P* approaches are substantially better than *D-RNS-Concat*, because the arithmetical computation processes of *S-RNS-Logic-P* methods are less complicated. The *S-RNS-Logic-P* with *SD-Post-Mul* is slightly worse than *S-RNS-Logic-P* with *SD-Pre-Mul* due to the more cycles required for the *SD-Post-Mul* algorithms. Compared to *D-RNS-Concat*, on average, the *S-RNS-Logic-P* with *SD-Pre-Mul* and *S-RNS-Logic-P* with *SD-Post-Mul* can achieve *30.3%* and *29.1%* execution time reduction, respectively.

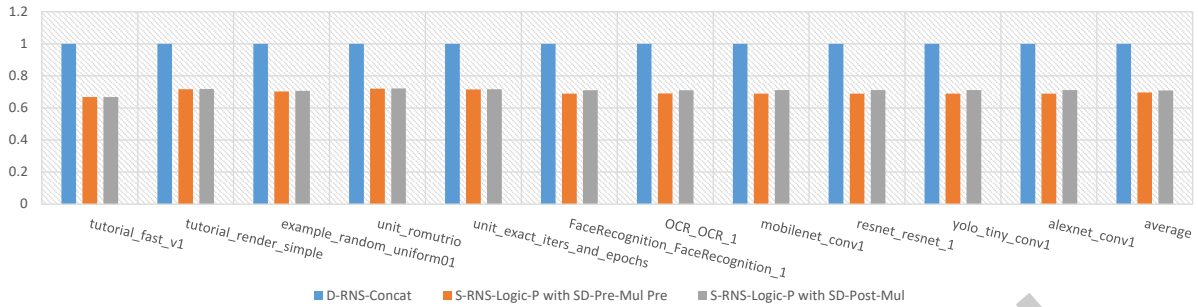


Fig. 16. Latency Evaluation of RNS Fixed-point Encoding; The y-axis values are normalized, where the latency values of *D-RNS-Concat* are normalized as 1.

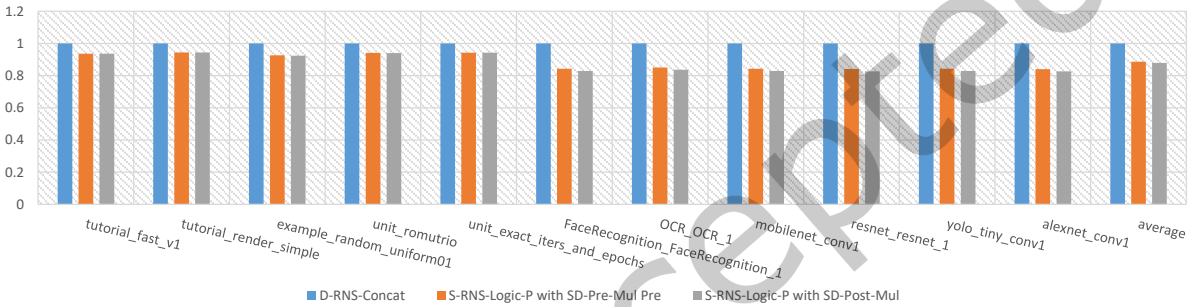


Fig. 17. Energy Evaluation of RNS Fixed-point Encoding; The y-axis values are normalized, where the energy values of *D-RNS-Concat* are normalized as 1.

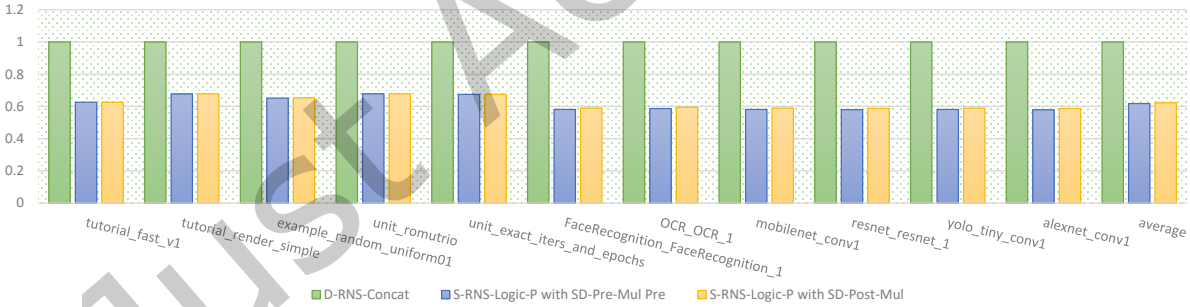


Fig. 18. Energy-delay Product (EDP) Evaluation of RNS Fixed-point Encoding; The y-axis values are normalized, where the EDP values of *D-RNS-Concat* are normalized as 1.

5.4 Energy Measurement

Energy is also crucial in measuring the computational efficiency from large-scale high-performance computers to lightweight IoT/edge devices. Figure 17 summarized the normalized energy consumptions of three RNS fixed-point methodologies. Similar to the latency results, *D-RNS-Concat* is worse than the two *S-RNS-Logic-P* methods. For the two *S-RNS-Logic-P* schemes, the one with *SD-Post-Mul* is slightly better. Compared to *D-RNS-Concat*, on average, the *S-RNS-Logic-P* with *SD-Pre-Mul* and *S-RNS-Logic-P* with *SD-Post-Mul* can reduce *11.4%* and *12.2%* energy, respectively.

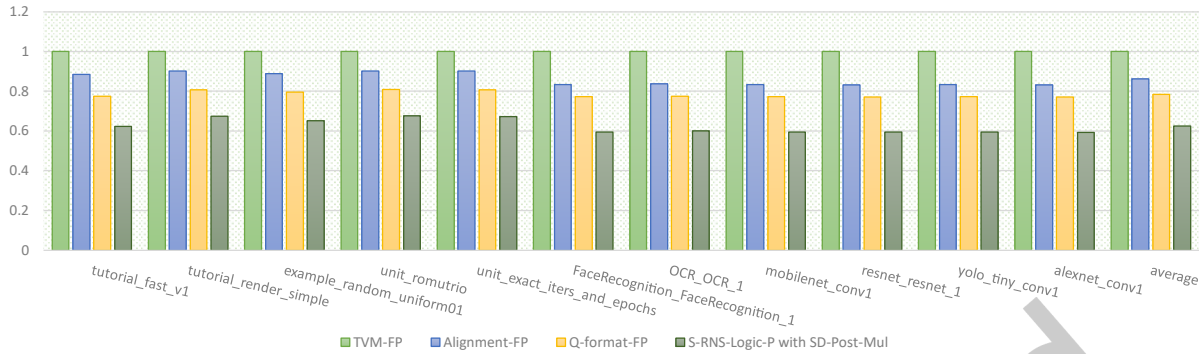


Fig. 19. Energy-delay Product (EDP) Comparisons Between Binary and RNS Fixed-point Encodings; The y-axis values are normalized, where the EDP values of *D-RNS-Concat* are normalized as 1.

5.5 Energy-delay Product (EDP) Measurement

Energy-delay Product (EDP) is a composite evaluation metric that integrates the consideration of latency and energy consumption. The smaller EDP value is better. Figure 18 shows the normalized EDP values of three RNS fixed-pointed methodologies. On average, the EDP values of *S-RNS-Logic-P* with *SD-Pre-Mul* and *S-RNS-Logic-P* with *SD-Post-Mul* obtain *38.2%* and 37.7% reduction from that of *D-RNS-Concat*, which indicates that the comprehensive computing efficiencies of two *S-RNS-Logic-P* schemes are very similar. However, if we combine the accuracy consideration from Section 5.2, *S-RNS-Logic-P* with *SD-Post-Mul* should be a better option.

5.6 EDP Comparisons Between Binary and RNS Fixed-point Encodings

Fixed-point strategies can be implemented in regular binary encoding. Q-format-FP [62], Alignment-FP [45], and TVM-FP [71] are three binary fixed-point schemes that will be further discussed in Section 6. For RNS encoding, according to our previous accuracy and EDP results in this section, *S-RNS-Logic-P* with *SD-Post-Mul* is the better solution. To make results more straightforward, we compare *S-RNS-Logic-P* with *SD-Post-Mul* with the three binary fixed-point encodings, as demonstrated in Figure 19. The EDPs of TVM-FP are normalized as 1. Compared with the Q-format-FP, which is observed best EDP from the three binary fixed-point encodings, *S-RNS-Logic-P* with *SD-Post-Mul* has a 20.5% EDP reduction on average.

5.7 EDP Comparisons of System-on-chip (SoC) with Processing-in Memory (PIM)

The improvements highly depend on the system architectures and which components we will apply the RNS fixed-point encoding. E.g, if we only utilize the RNS strategy in a single CPU ALU, then the overall improvement rate may be limited because the data movement between memory and computational units should contribute to the majority of the total energy [15]. One potential application of RNS fixed-point encoding is a SoC with processing-in memory (PIM), which can significantly reduce data movement. We model an RNS-PIM system similar to RNSnet [60] but with our fixed-point encoding (*S-RNS-Logic-P* with *SD-Post-Mul*). The results of PIM with regular binary computational units are normalized as 1. Because the silicon area of the single RNS adder/multiplier is less than the regular binary adder/multiplier, we integrate more RNS adder/multiplier into memory to ensure the RNS computational unit area is close but no more than the area of the binary version. In other words, RNS architecture with fixed-point encoding provides better computing powers and, therefore, lowers the response time. The full system EDP comparison between the SoC with binary-PIM and the SoC with RNS-PIM is illustrated in Figure 20. Compared with the binary-PIM, on average, our RNS fixed-point encoding can reduce the full system EDP by 18.7%.

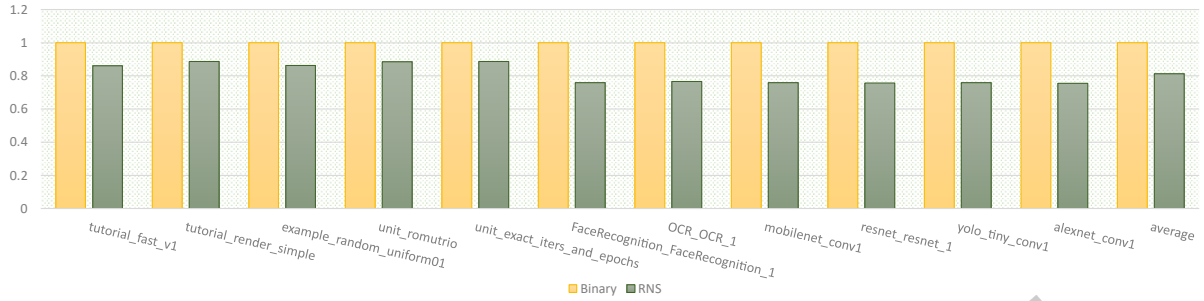


Fig. 20. EDP Comparisons between the SoC with Binary-PIM and the SoC with RNS-PIM; The y-axis values are normalized, where the EDP values of *Binary* are normalized as 1.

6 RELATED WORK

The real number encoding and associated arithmetical algorithms are critical factors that determine the efficiency of the underline system, especially in serving addition/multiplication-intensive applications, such as training/inference of various AI models [26, 51, 67]. RNS is a powerful and low-cost technique [64] for integer applications with a high ratio of addition/subtraction/multiplication instructions. Appropriate RNS real number encoding could be one of the strategies to further raise the efficiency of AI accelerators (e.g., Google TPU [40, 41], Samsung NPU [38, 54], Tesla FSD Chip [11, 66], Nvidia NVDLA [25, 28], Arm MLP [16]).

The IEEE 754 Floating-Point [1, 3] is a common standard for the current binary computers. The RNS floating-point encoding is a relatively straightforward method to represent real numbers. Chiang etc. [39] provided an RNS floating-point method that not only may require calling RNS scaling-down function (scale down by 10) multiple times to normalize the mantissae but also need many extra requirements for RNS moduli, such as need a *1-bit* redundant residue for each RNS number to quickly verify its parity, and no modulus is an even number. The RNS floating-point methods from Ghosh etc. [31] also consist of RNS unfriendly subroutines such as RNS min/max comparisons and general radix right shiftings. The method proposed by Dhanabal etc. [24] may need several RNS/Binary conversions in each multiplication. Even though optimizations exist in the above floating-point schemes, they are still inefficient due to the expensive algorithms involved, eliminating the default benefits of RNS encoding.

Fixed-point encoding is another option for the real number representation, which typically loses precision and range to trade other benefits, including less power and better performance. Saokar et al. [62] utilize the Q-format fixed-point representation with the multiplier optimization via the Urdhava Tiryakbhyam method. Essentially, they ensure the computation is the same as integer arithmetic calculation to reduce time, area, and power overhead. However, this approach only focuses on Q15 and Q31 formats, which can only represent fractional numbers instead of real numbers. Lee et al. [45] allow users to define fixed-point data types such as *fixed_point<w,e>*, where *w* indicates the width, and *e* is for the exponent part. This fixed-point representation may process like regular integer computation. However, operand alignment may require in addition or subtraction operations, which is considered as extra overhead compared with traditional integer computation. Yang et al. [71] extend a fixed-point type on the TVM compiler. This fixed-point type essentially is a 16-bit integer and is used to replace a 32-bit floating point to reduce the energy overhead. However, their method requires converting floating point operands to fixed-point before the convolution and finally converting the output back to floating point, which further increases the computational overhead. Besides overcoming the previously mentioned limitations from [62] , [45], and [71], even for the advantage that is similar to binary integer computation, our method is optimized one step further. We not only ensure the integer computational benefits but also integrate the RNS strategy to explore the delay, power, and silicon area benefits further.

For some applications, relatively less precision and a smaller range could still can ensure correctness. Olsen proposed a fixed-point RNS multiplication [53] and demonstrated 7-9 times more efficient compared with a binary matrix multiplier. However, this algorithm still needs to include some relatively high overhead subroutines, such as a division, base-extension [33, 69], and mixed-radix conversions [21, 29, 36]. So in this paper, we explore the RNS fixed-point encodings and associated multiplication algorithms to better support the architecture design of future AI accelerators.

7 CONCLUSION

With the explosive growth of the AI community, training state-of-the-art AI models efficiently becomes a pressing challenge. The colossal training overhead of large-scale models may impede the continuous evolution of AI techniques. This paper presents a fixed-point data encoding to assist efficient AI accelerator design, which smoothly obtains the low-cost benefits from Residue Number Systems (RNS). According to experimental results, our *S-RNS-Logic-P with SD-Post-Mul* approach gets *37.7%* Energy-delay Product (EDP) reduction compared to the straightforward *D-RNS-Concat* scheme while holding good accuracy. For *D-RNS-Concat* and *S-RNS-Logic-P with SD-Pre-Mul* methods, we extend implementation details in this paper. To further facilitate the development of an RNS-based AI accelerator, we also present architectures of fixed-pointed multipliers for these three RNS real data presentations. For accuracy, we mainly focus on the evaluations of single arithmetic operations in this paper. Systematically evaluating the accuracies of neural networks based on three RNS fixed-point methods will be one of our main future works. Although the primary objective of our *S-RNS-Logic-P* with *SD-Post-Mul* method is to raise the efficiency of AI accelerators, this approach can also be extended to error-tolerant domains (e.g., error correction of quantum computing) via attaching RNS redundant residues.

ACKNOWLEDGMENTS

We truly appreciate all the anonymous reviewers for their time to evaluate this work.

REFERENCES

[1] 2008. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935
[2] 2016. DeepBench. https://github.com/baidu-research/deepbench.
[3] 2019. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229
[4] 2019. nanobench. https://github.com/martinus/nanobench.
[5] 2023. TOP500 LIST - JUNE 2023. https://www.top500.org/lists/top500/list/2023/06/.
[6] opengenus.org. Challenges of Quantization in Machine Learning (ML). https://iq.opengenus.org/challenges-of-quantization/.
[7] Gerardo Adesso. 2022. GPT4: The ultimate brain. Authorea Preprints (2022).
[8] Ayaz Akram and Lina Sawalha. 2019. Validation of the gem5 Simulator for x86 Architectures. In 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). 53–58. https://doi.org/10.1109/PMBS49563.2019.00012
[9] Pietro Albicocco, Gian Carlo Cardarilli, Alberto Nannarelli, and Marco Re. 2014. Twenty years of research on RNS for DSP: Lessons learned and future perspectives. In 2014 International Symposium on Integrated Circuits (ISIC). 436–439. https://doi.org/10.1109/ISICIR.2014.7029575
[10] P. V. Ananda Mohan. 2016. RNS in Cryptography. Springer International Publishing, Cham, 263–347. https://doi.org/10.1007/978-3-319-41385-3_10
[11] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. Computer and redundancy solution for the full self-driving computer. In 2019 IEEE Hot Chips 31 Symposium (HCS). IEEE Computer Society, 1–22.
[12] Keren Bergman, Tom Conte, Al Gara, Maya Gokhale, Mike Heroux, Peter Kogge, Bob Lucas, Satoshi Matsuoka, Vivek Sarkar, and Olivier Temam. 2019. Future high performance computing capabilities: Summary report of the advanced scientific computing advisory committee (ascac) subcommittee. Technical Report. USDOE Office of Science (SC)(United States).
[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. SIGARCH Comput. Archit. News 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718
[14] Kwame Osei Boateng. 2009. Bioinformatic: An Important Application Area of Residue Number System. In International Conference on Mathematics and its Applications.

[15] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. 2018. Google workloads for consumer devices: Mitigating data movement bottlenecks. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 316–331.
[16] Ian Bratt. 2018. Arm's first-generation machine learning processor. In Hot Chips.
[17] Reuben D Budiardja, Mark Berrill, Markus Eisenbach, Gustav R Jansen, Wayne Joubert, Stephen Nichols, David M Rogers, Arnold Tharrington, and OE Bronson Messer. 2023. Ready for the Frontier: Preparing Applications for the World's First Exascale System. In International Conference on High Performance Computing. Springer, 182–201.
[18] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. 2012. Accuracy evaluation of GEM5 simulator system. In 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC). 1–7. https://doi.org/10.1109/ReCoSoC.2012.6322869
[19] Gian Carlo Cardarilli, Alberto Nannarelli, and Marco Re. 2017. RNS Applications in Digital Signal Processing. Springer International Publishing, Cham, 181–215. https://doi.org/10.1007/978-3-319-49742-6_8
[20] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2019. Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28, 2 (2019), 530–543.
[21] NB Chakraborti, John S. Soundararajan, and A. L. Narasimha Reddy. 1986. An implementation of mixed-radix conversion for residue number applications. IEEE Transactions on computers 35, 08 (1986), 762–764.
[22] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, Ilya Sutskever. 2018. AI and Compute. https://openai.com/blog/ai-and-compute/.
[23] Bobin Deng. 2021. Scalable Energy-efficient Microarchitectures with Computational Error Tolerance. In in Georgia Institute of Technology.
[24] R Dhanabal, V Barathi, Sarat Kumar Sahoo, Naamatheertham R Samhitha, Neethu Acha Cherian, and Pretty Mariam Jacob. 2014. Implementation of floating point MAC using residue number system. In 2014 International Conference on Reliability Optimization and Information Technology (ICROIT). IEEE, 461–465.
[25] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim. In 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2). 21–25. https://doi.org/10.1109/EMC249363.2019.00012
[26] Alhussein Fawzi, Matej Balog, and Aja et al. Huang. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. Nature 610, 7930 (2022), 47–53.
[27] Michael Feldman. 2011. Powering Up Exascale. https://www.hpcwire.com/2011/09/01/powering_up_exascale/.
[28] Karl Freund. 2018. Arm Chooses NVIDIA Open-Source CNN AI Chip Technology. In Forbes. Moor Insights and Strategy.
[29] Kazeem Alagbe Gbolagade and Sorin Dan Cotofana. 2009. An O(n) Residue Number System to Mixed Radix Conversion technique. In 2009 IEEE International Symposium on Circuits and Systems. 521–524. https://doi.org/10.1109/ISCAS.2009.5117800
[30] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. 2021. AI and Memory Wall. RiseLab Medium Post (2021).
[31] Aniruddha Ghosh, Satrughna Singha, and Amitabha Sinha. 2012. " Floating point RNS" a new concept for designing the MAC unit of digital signal processor. ACM SIGARCH Computer Architecture News 40, 2 (2012), 39–43.
[32] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 13–22. https://doi.org/10.1109/ISPASS.2014.6844457
[33] C. W. Hastings. 1966. Automatic detection and correction of errors in digital computers using residue arithmetic. In Region Six Annu. Conf., IEEE. 429–464.
[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.
[35] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017).
[36] CH Huang. 1983. A fully parallel mixed-radix conversion algorithm for residue number applications. IEEE Transactions on computers 32, 04 (1983), 398–402.
[37] David Jacquemin, Ahmet Can Mert, and Sujoy Sinha Roy. 2022. Exploring RNS for Isogeny-based Cryptography. Cryptology ePrint Archive (2022).
[38] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, et al. 2021. Sparsity-aware and re-configurable NPU architecture for Samsung flagship mobile SoC. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 15–28.

[39] Jen-Shiun Chiang and Mi Lu. 1991. Floating-point numbers in residue number systems. Computers & Mathematics with Applications 22, 10 (1991), 127–140. https://doi.org/10.1016/0898-1221(91)90200-N
[40] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and Evaluation of the First Tensor Processing Unit. IEEE Micro 38, 3 (2018), 10–19. https://doi.org/10.1109/MM.2018.032271057
[41] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th annual international symposium on computer architecture. 1–12.
[42] Felix Kaiser, Stefan Kosnac, and Ulrich Brüning. 2019. Development of a RISC-V-conform fused multiply-add floating-point unit. Supercomputing Frontiers and Innovations 6, 2 (2019), 64–74.
[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc.
[44] Chao-Lin Lee, Min-Yih Hsu, Bing-Sung Lu, Ming-Yu Hung, and Jenq-Kuen Lee. 2020. Experiment and enabled flow for GPGPU-Sim simulators with fixed-point instructions. Journal of Systems Architecture 111 (2020), 101783. https://doi.org/10.1016/j.sysarc.2020.101783
[45] Chao-Lin Lee, Min-Yih Hsu, Bing-Sung Lu, Ming-Yu Hung, and Jenq-Kuen Lee. 2020. Experiment and enabled flow for GPGPU-sim simulators with fixed-point instructions. Journal of Systems Architecture 111 (2020), 101783.
[46] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv:2306.00978 [cs.CL]
[47] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. 2020. FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 29, 4 (2020), 774–787.
[48] Stephen McAleese. 2023. Retrospective on 'GPT-4 Predictions' After the Release of GPT-4. https://www.lesswrong.com/posts/iQx2eeHKLwgBYdWPZ/ retrospective-on-gpt-4-predictions-after-the-release-of-gpt/.
[49] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 54), Aarti Singh and Jerry Zhu (Eds.). PMLR, 1273–1282. https://proceedings.mlr.press/v54/mcmahan17a.html
[50] Hiroki Nakahara and Tsutomu Sasao. 2018. A High-speed Low-power Deep Neural Network on an FPGA based on the Nested RNS: Applied to an Object Detector. In 2018 IEEE International Symposium on Circuits and Systems (ISCAS). 1–5. https://doi.org/10.1109/ISCAS.2018.8351850
[51] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC). 32–41. https://doi.org/10.1109/HiPC.2018.00013
[52] Olatunbosun Lukumon Olawale, Lawal Tunde Dauda, and Gbolagade Kazeem Alagbe. 2019. An Efficient RNS Arithmetic in Bioinformatics sequences. In International Journal of Computer Science Issues (IJCSI), Vol. 16. 19–26.
[53] Eric B. Olsen. 2018. RNS Hardware Matrix Multiplier for High Precision Neural Network Acceleration: "RNS TPU". In 2018 IEEE International Symposium on Circuits and Systems (ISCAS). 1–5. https://doi.org/10.1109/ISCAS.2018.8351352
[54] Jun-Seok Park, Heonsoo Lee, Dongwoo Lee, Jewoo Moon, Suknam Kwon, SangHyuck Ha, MinSeong Kim, Junghun Park, Jihoon Bang, and Sukhwan Lim Inyup Kang. 2021. Samsung Neural Processing Unit : An AI accelerator and SDK for flagship mobile AP. In 2021 IEEE Hot Chips 33 Symposium (HCS). 1–21. https://doi.org/10.1109/HCS52781.2021.9567119
[55] Senthil Pitchai and Saravanan Pitchai. 2023. Area-latency efficient floating point adder using interleaved alignment and normalization. Microprocessors and Microsystems 99 (2023), 104842.
[56] A.P. Preethy and D. Radhakrishnan. 1999. A 36-bit balanced moduli MAC architecture. In 42nd Midwest Symposium on Circuits and Systems (Cat. No.99CH36356), Vol. 1. 380–383 vol. 1. https://doi.org/10.1109/MWSCAS.1999.867285
[57] A.P. Preethy and Dayalu Radhakrishnan. 2000. RNS-based logarithmic adder. Computers and Digital Techniques, IEE Proceedings - 147 (08 2000), 283 – 287. https://doi.org/10.1049/ip-cdt:20000529
[58] V Rajaraman. 2023. Frontier—World's First ExaFLOPS Supercomputer. Resonance 28, 4 (2023), 567–576.
[59] Pramod V Rampur, M Jagadish, and G Yogeesha. 2016. Design and implementation of advanced array multiplier for binary multiplication on FPGA. International Journal of Engineering Research & Technology (2016), 142–144.
[60] Sahand Salamat, Mohsen Imani, Sarangh Gupta, and Tajana Rosing. 2018. RNSnet: In-Memory Neural Network Acceleration Using Residue Number System. In 2018 IEEE International Conference on Rebooting Computing (ICRC). 1–12. https://doi.org/10.1109/ICRC.2018.8638592
[61] Katharine Sanderson. 2023. GPT-4 is here: what scientists think. Nature 615, 7954 (2023), 773.
[62] Sandesh S Saokar, RM Banakar, and Saroja Siddamal. 2012. High speed signed multiplier for digital signal processing applications. In 2012 IEEE International Conference on Signal Processing, Computing and Control. IEEE, 1–6.

[63] Sandesh S. Saokar, R. M. Banakar, and Saroja Siddamal. 2012. High speed signed multiplier for Digital Signal Processing applications. In 2012 IEEE International Conference on Signal Processing, Computing and Control. 1–6. https://doi.org/10.1109/ISPCC.2012.6224373
[64] Sriseshan Srikanth, Paul G. Rabbat, Eric R. Hein, Bobin Deng, Thomas M. Conte, Erik DeBenedictis, Jeanine Cook, and Michael P. Frank. 2018. Memory System Design for Ultra Low Power, Computationally Error Resilient Processor Microarchitectures. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). 696–709. https://doi.org/10.1109/HPCA.2018.00065
[65] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. 2020. Degree-quant: Quantization-aware training for graph neural networks. arXiv preprint arXiv:2008.05000 (2020).
[66] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, et al. 2020. Compute solution for tesla's full self-driving computer. IEEE Micro 40, 2 (2020), 25–35.
[67] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel Multi Channel convolution using General Matrix Multiplication. In 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP). 19–24. https://doi.org/10.1109/ASAP.2017.7995254
[68] Yap June Wai, Zulkalnain Mohd Yussof, Sani Irwan Md. Salim, and Lim Kim Chuan. 2018. Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA. International Journal of Advanced Computer Science and Applications 9 (2018).
[69] R. W. Watson. 1965. Error detection and correction and other residue interacting operations in a redundant residue number system. In in University of California, Berkeley.
[70] Chun-Chieh Yang, Yi-Ru Chen, Hui-Hsin Liao, Yuan-Ming Chang, and Jenq-Kuen Lee. 2023. Auto-Tuning Fixed-Point Precision with TVM on RISC-V Packed SIMD Extension. ACM Trans. Des. Autom. Electron. Syst. 28, 3, Article 33 (mar 2023), 21 pages. https://doi.org/10.1145/3569939
[71] Chun-Chieh Yang, Yi-Ru Chen, Hui-Hsin Liao, Yuan-Ming Chang, and Jenq-Kuen Lee. 2023. Auto-tuning Fixed-point Precision with TVM on RISC-V Packed SIMD Extension. ACM Transactions on Design Automation of Electronic Systems 28, 3 (2023), 1–21.
[72] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2020. Manticore: A 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing. IEEE Micro 41, 2 (2020), 36–42.

Received 22 June 2023; revised 21 March 2024; accepted 30 April 2024