<table>
<thead>
<tr>
<th>Title</th>
<th>Simple, fast, and exact RNS scaler for the three-moduli set {2n - 1, 2n, 2n + 1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Chang, Chip-Hong; Low, Jeremy Yung Shern</td>
</tr>
<tr>
<td>Citation</td>
<td>Chang, C.-H., &amp; Low, J. Y. S. (2011). Simple, fast, and exact RNS scaler for the three-moduli set {2n - 1, 2n, 2n + 1}. IEEE transactions on circuits and systems I : regular papers, 58(11), 2686-2697.</td>
</tr>
<tr>
<td>Date</td>
<td>2011</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10220/25604">http://hdl.handle.net/10220/25604</a></td>
</tr>
<tr>
<td>Rights</td>
<td>© 2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [<a href="http://dx.doi.org/10.1109/TCSI.2011.2142950">http://dx.doi.org/10.1109/TCSI.2011.2142950</a>].</td>
</tr>
</tbody>
</table>
Simple, Fast and Exact RNS Scaler for The Three-Moduli Set \{2^n-1, 2^n, 2^n+1\}

Chip-Hong Chang, Senior Member, IEEE and Jeremy Yung Shern Low

Abstract—Scaling in RNS has always been conceived as a performance bottleneck similar to the residue-to-binary conversion problem due to the inefficient inter-modulo operation. In this paper, a simple and fast scaling algorithm for the three-moduli set \{2^n-1, 2^n, 2^n+1\} RNS is proposed. The complexity of inter-modulo operation has been resolved by a new formulation of scaling an integer in RNS domain by one of its moduli. By elegant exploitation of the Chinese Remainder Theorem and the number theoretic properties for this moduli set, the design can be readily implemented by a standard cell based design methodology. The low cost VLSI architecture without any read-only memory (ROM) makes it easier to fuse into and pipeline with other residue arithmetic operations of a RNS-based processor to increase the throughput rate. The proposed RNS scaler possesses zero scaling error and has a critical path delay of only $2[\log n]+9$ units in unit-gate model. Besides the scaled residue numbers, the scaled integer in normal binary representation is also produced as a byproduct of this process, which saves the residue-to-binary converter when the binary representation of scaled integer is also required. Our experimental results show that the proposed RNS scaler is smaller and faster than the most area-efficient adder-based design and the fastest ROM-based design besides being the most power efficient among all scalers evaluated for the same three-moduli set.

Index Terms—Chinese Remainder Theorem, full-adder-based architecture, inner product step processor, residue number system, scaling.

I. INTRODUCTION

The new generation of portable electronic devices, such as camcorders, digital cameras, 3G cell phone, etc. have incorporated many sophisticated real-time application-specific digital signal processing (DSP) functions. It has become increasingly difficult to keep up with the logical culmination of the demands for higher performance and lower power budget with the increased versatility of electronic products. Residue Number System (RNS) springs up as an alternative number system of special advantage for mapping complex DSP algorithms into application-specific hardware accelerators. This is due to the fact that long integer addition, subtraction and multiplication operations can be decomposed into smaller carry-free modulo operations for parallel processing. By introducing subword-level parallelism into an algorithm, RNS can tolerate a larger reduction in supply voltage than the corresponding weighted binary number system architecture for a given delay specification. The supply voltage reduction results in a quadratic power reduction and facilitates more efficient exploitation of multi-supply and multi-threshold voltage scaling techniques to further reduce the total static and dynamic power dissipation [1]–[2]. The usefulness of RNS has been demonstrated especially in the design of inner product step processor (IPSP) like architectures [3]–[7]. The most recent study [7] based on the two’s complement and the mixed RNS-binary architecture implementations of an adaptive channel equalization filter with a large number of taps for wireless communications showed that the latter offer remarkable savings in area and power consumption without any degradation in performance. The reason for using a hybrid-RNS in [7] instead of a true RNS system is to avoid the use of complex and expensive scaling circuits.

Reverse conversion of an integer from its residue number representation [8]–[13], [43] and scaling of an integer number by a known constant [14]–[25] are two of the most important problems in RNS. Scaling is of particular significance for implementing IPSP and iterative digital processing systems dominated by a large number of additions and multiplications because it ensures the computed results of the preceding stages do not exceed the dynamic range of the system. Being inefficient in inter-modular operations, implementing scaling operation in RNS domain entices considerably long delay and high hardware complexity, as in the reverse conversion. Unlike the reverse conversion, which is required only when the final result is needed to be communicated with a normal binary number system, scaling operations are executed repeatedly in IPSP-like architectures to avoid the severe consequence of overflow in modulo arithmetic. In [7], adaptation is carried out in the binary domain to avoid the speed bottleneck of RNS scaling, but this is viable provided that the channel variations in time are slower with respect to the data rate.

To avoid overflow error in iterative computations, a RNS scaler should be placed after each RNS computation stage (each stage may involve one or more arithmetic operations), as shown in Fig. 1(a). RNS scaling problem is typically formulated based on modulo reduction of the least integer function of division or by Chinese Remainder Theorem (CRT). The former problem is generally solved by computationally intensive base extension methods [15], [17], [19], [23]–[25] and the outputs are free from
scaling error. On the other hand, CRT-based approaches [14, 16], [20]-[22] usually produce scaled integer with some fractional error. The error could escalate and might not be tolerable for critical applications. All the aforementioned RNS scaling algorithms, except [19], [24] and [25], are implemented using ROMs. ROM-based approaches have low throughput. Although ROM matrices can be replaced with modulo multipliers, the cost is unacceptably high, especially for large moduli. In the absence of an efficient RNS scaler, a binary scaler may be used in a hybrid RNS but it must be preceded by an expensive reverse converter, as shown in Fig. 1(b). It is therefore imperative to have a high-speed scaler that operates directly in the RNS domain with lower complexity than a RNS-to-binary converter. Given that scaling in RNS is an inherently inter-modular operation whereby knowledge of all of the residue digits is essential to produce the correct final result [15], [20], such a problem is better tackled with a moduli set that possesses good number theoretic properties.

This paper presents a new and simple adder-based RNS scaler for the three-moduli set \( \{2^n - 1, 2^n, 2^n + 1\} \). The choice of this RNS for overcoming the scaling problem is well justified by the relative simplicity and efficiency in implementing modulo addition, multiplication and shifter circuits for these three moduli proven by the humongous amount of research publications [26]-[34]. The conversion from the normal binary representation to residues for moduli of types \( 2^n \) and \( 2^n \pm 1 \) is straightforward and the reverse conversion problem for this special three-moduli set has been well solved [8], [9]. Most importantly, the inputs to our proposed RNS scaler are in residue form, and the scaled output can be produced efficiently in both normal binary and residue forms. This has an additional advantage of eliminating the reverse converter in the last stage of the architecture shown in Fig. 1(a).

The paper is organized as follows: recent works on RNS scaling operation are reviewed in Section II, followed by the derivation of the new RNS scaling algorithm and its error analysis in Section III. The architecture of the proposed scaler is presented in Section IV. Section V analyzes the area and time complexity of its implementation, and compares the synthesis results with other scaler designs. The paper is concluded in Section VI.

II. RELATED WORK

The earliest research on RNS scaling problem dated back to the sixties whence scaling an integer by a product of any subset of the moduli in RNS can be performed in \( n \) clock cycles [10]. Since then, many new techniques have been proposed in the literature. This section reviews some recent developments on RNS scaling algorithms that have an impact on hardware implementation.

Shenoy and Kumaresan [14] proposed a scaling technique based on the CRT. In their work, a redundant residue digit is introduced to enable the parallel computation of the scaled residues, and a novel decomposition of CRT is applied to reduce the maximum scaling error to unity. In [21], Griffin et al. first used the least integer function \([\cdot]\) in CRT to produce the scaled output with error bounded by the number of modulus \( L \) used in the RNS (hence, it is known as L-CRT). Later, a generalization of L-CRT, called L(\( \epsilon + \delta \)) -CRT [20] is proposed to provide the flexibility in choosing the scaling factor but the new reduced system modulus comes at a cost of potentially large errors. The catastrophic error band may be avoided with carefully selected reduced modulus for a pre-specified scaling factor. Another scaling algorithm that utilizes CRT was put forward by Ulman et al. [22]. The latter scaling method could be considered as an improved version of the design in [10], as it eliminates the use of redundant modulus without imposing any restrictions on the system moduli and on the scaling factor. Besides, the scaling error of this algorithm is not greater than 1.5. The aforementioned scaling algorithms use CRT in such a way that the residue representation of an integer in RNS is partially converted to its binary representation before producing the scaled output. Furthermore, they are ROM-based design which cannot be easily pipelined for high-throughput applications.

More recent works on scaling an integer, \( X \) by a factor \( k \) for the moduli set \( \{m_1, m_2, \ldots, m_n\} \) is based on the scaling operation defined in [17]. It states that if \( k \) is the scaling constant and \( Y \) is the result of scaling \( X \) by \( k \), then \( X = kY + \lfloor X \rfloor_k \), where \( \lfloor X \rfloor_k \) denotes the remainder of \( X \) divided by \( k \). Thus,
Barsi and Pinotti [15] derived a scaling algorithm by applying the base extension to (1) without approximation. The latter work has once again manipulated the CRT so that their design can perform both base extension and exact scaling without any system redundancy. Using almost similar scaling algorithm, Garcia and Lloris [17] had proposed a different look-up scheme, which leads to a faster and simpler implementation of the RNS scaling operation than [15]. The drawback is its restriction on the number of moduli in the system—significantly large amount of memory is required for a large number of moduli. Similarly, these two designs also use memory for implementation. All these memory-based designs suffer from poor pipelinability and dramatic increase in hardware cost with the number of system moduli. Recently, Meyer-Base and Stourasitis [18] had come out with a scaling algorithm for small word length applications. It does not rely on a conversion back to a weighted number system. Instead, it extended the Division Remainder Zero Theorem of Szabo and Tanaka [10] to perform scaling directly on the residue digits.

The first and only full adder (FA) based exact scaling algorithm that operates directly on the residue digits was proposed by Dasygenis et al. [19]. It uses the axiom,

\[ |X - X|_m = |x - X|_m \]

(1)

where \( x_i = |X|_m \) is a residue of the RNS defined by the moduli set \( \{m_1, m_2, \ldots, m_N\} \). The computation is divided into several major stages as shown in Fig. 2. Each stage can be implemented with only FAs, which makes the hardware cost remarkably lower and throughput rate significantly higher than all the previous designs. Unfortunately, this algorithm has ignored the fact that \( |X|_m \) is an inter-modular operation and it cannot be simply resolved by treating \( X \) as an overflowed digit or a residue over each modulus independently. The authors of [19] also made a catastrophic assumption that the scaling constant, \( k \), can be chosen to be a product of various moduli, i.e., \( k = \prod_{i=1}^{S} m_i \) with \( S < N \), but it has been proven that multiplicative inverse of \( k \) for \( m_i \) \((1 \leq i \leq S)\) does not exist [15, 17]. Consequently, the output \( (y_1, y_2, \ldots, y_N) \) calculated by the algorithm of [19] is not a valid RNS representation of scaling \( X \) by \( k \). Using the first example from [19], this error is illustrated by the numerical calculation in Table I. The actual residue digits of the scaled number \((254, 204, 154)\) are different from those calculated \((4, 3, 3)\) using the scaling method in [19]. In fact, the condition \((k, m_i) = 1\) for the existence of \( k^{-1} \) is not met because of \((k, m_1) = (255, 255) = 255\). Besides, it may not be possible to access to the already overflowed digits because the modulo operators in RNS are usually designed such that modulo reductions are performed to keep the final results and even their intermediate computations within the legitimate ranges of the respective moduli.

In view of the above, the only valid adder-based RNS scaler design technique that utilizes (1) was proposed in [24] and [25]. The scaler of [24] is dedicated to the three moduli set \( \{2^w - 1, 2^w, 2^w + 1\} \) with scaling constant \( 2^w \) while [25] has extended it to general moduli sets with the proviso that the moduli are coprime with the scaling constant. The generalization has restricted its hardware simplification, leading to increased hardware cost and reduced performance over those of [24].

**TABLE I: COMPUTATION OF ACTUAL RESIDUE DIGITS OF SCALED NUMBER**

<table>
<thead>
<tr>
<th>Modulus, ( m_i )</th>
<th>255</th>
<th>256</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic range, ( M )</td>
<td>16776960</td>
<td>65792</td>
<td>65535</td>
</tr>
<tr>
<td>( M_i )</td>
<td>128</td>
<td>255</td>
<td>129</td>
</tr>
<tr>
<td>( M_i \cdot M_i^{-1} )</td>
<td>8421376</td>
<td>16711425</td>
<td>8421120</td>
</tr>
<tr>
<td>overflowed ( x_i = x_i' )</td>
<td>1100</td>
<td>900</td>
<td>800</td>
</tr>
<tr>
<td>( x_i =</td>
<td>x_i</td>
<td>_{m_i} )</td>
<td>80</td>
</tr>
<tr>
<td>( x_i \cdot M_i^{-1} \cdot M_i^{-1} )</td>
<td>673710080</td>
<td>2205908100</td>
<td>244212480</td>
</tr>
<tr>
<td>Integer ( \hat{X} )</td>
<td>3316100</td>
<td>13004</td>
<td>13004</td>
</tr>
<tr>
<td>Scaler ( k )</td>
<td>255</td>
<td>256</td>
<td>257</td>
</tr>
<tr>
<td>( y = [X/k] )</td>
<td>13004</td>
<td>13004</td>
<td>13004</td>
</tr>
<tr>
<td>( x_i )</td>
<td>254</td>
<td>204</td>
<td>154</td>
</tr>
</tbody>
</table>

**III. PROPOSED SCALING ALGORITHM**

### A. Preliminaries

In RNS, an integer \( X \) is represented by an \( N \)-tuple \((x_1, x_2, \ldots, x_N)\) with respect to a set of pairwise relatively prime numbers \((m_1, m_2, \ldots, m_N)\), where \( x_i = |X|_{m_i}, i = 1, 2, \ldots, N \) and \( |X|_m \) is defined as \( X \mod m \). The dynamic range of a selected
moduli set \( \{m_1, m_2, \ldots, m_n\} \) is given by
\[
M = \prod_{i=1}^{N} m_i
\]  
(2)

Based on CRT [14], \( X \) is related to its residue digits by
\[
X = \left\lfloor \sum_{i=1}^{N} M_i \left\lfloor \frac{x}{m_i} \right\rfloor \right\rfloor
\]  
(3)

where \( M_i = \frac{M}{m_i} \) and \( \left\lfloor M_i^{-1} \right\rfloor \) is the multiplicative inverse of \( \left\lfloor M_i^{-1} \right\rfloor \).

Our proposed scaling algorithm is designed for the three-moduli set \( \{2^n - 1, 2^n, 2^n + 1\} \), so \( N = 3 \) and (3) becomes
\[
X = \left| m_2m_3 \left\lfloor M_1^{-1} \right\rfloor m_1 x_1 + m_1m_3 \left\lfloor M_2^{-1} \right\rfloor m_2 x_2 + m_1m_2 \left\lfloor M_3^{-1} \right\rfloor m_3 x_3 \right|_M
\]  
(4)

The following axioms and theorems are used for the derivation of our scaling algorithm.

Axiom 1: \( A[X]_M = AX \) \( \left\lfloor \left\lfloor X \right\rfloor \right\rfloor \)

Axiom 2: \( \left\lfloor X + \frac{a}{b} \right\rfloor = \left\lfloor \left\lfloor X \right\rfloor + \frac{a}{b} \right\rfloor \)

Axiom 3: \( \left\lfloor X \cdot \frac{a}{b} \right\rfloor = \left\lfloor \left\lfloor X \right\rfloor \cdot \frac{a}{b} \right\rfloor \)

Theorem 1: Given \( p = kq \) where \( k \) is an integer, \( \left\lfloor \left\lfloor X \right\rfloor \right\rfloor = \left\lfloor X \right\rfloor \)

Proof: Based on the definition of modulo operation [14],
\[
\left\lfloor X \right\rfloor = X - \epsilon p, \epsilon = 0, 1, 2, \ldots
\]  
(5)

From Axioms 2 and 3,
\[
\left\lfloor \left\lfloor X \right\rfloor \right\rfloor = \left\lfloor X - \epsilon p \right\rfloor = \left\lfloor X \right\rfloor - \left\lfloor \epsilon p \right\rfloor
\]  
(6)

Since \( p \) is divisible by \( q \), \( \left\lfloor \epsilon p \right\rfloor = 0,
\[
\left\lfloor \left\lfloor X \right\rfloor \right\rfloor = \left\lfloor X \right\rfloor
\]  
(7)

Theorem 2[42]: Given the moduli set of \( \{m_1, m_2, m_3\} \) with \( m_1 = 2^n - 1, m_2 = 2^n \) and \( m_3 = 2^n + 1 \), the followings hold true.
\[
\left\lfloor M_1^{-1} \right\rfloor m_1 = 2^n - 1
\]  
(8)
\[
\left\lfloor M_2^{-1} \right\rfloor m_2 = 2^n - 1
\]  
(9)
\[
\left\lfloor M_3^{-1} \right\rfloor m_3 = 2^n + 1
\]  
(10)

B. New Formulation of RNS Scaling

Based on the above preliminaries, the following theorem is proposed to reduce the complexity of inter-modular operation of RNS scaling.

Theorem 3: Let \( x_i = \left\lfloor X \right\rfloor \) for \( i = 1, 2, 3 \) and \( m_1 = 2^n - 1, m_2 = 2^n \) and \( m_3 = 2^n + 1 \). If \( k = m_2 = 2^n \), then
\[
\left\lfloor X \right\rfloor = \left\lfloor x_1 - x_2 \right\rfloor
\]  
(11)

\[
\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor \left(2^{2^m-1} + 2^{2^m-2} \right) x_1 - 2^n x_2 + \left(2^{2^m-1} + 2^{2^m-2} - 1 \right) x_3 \right\rfloor
\]  
(12)
\[
\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor x_2 + 2^n x_3 \right\rfloor
\]  
(13)

The formulation of this theorem and its proof is given as follows.

By definition, scaling an integer variable \( X \) by a constant integer \( k \) can be obtained by dividing both sides of (4) by \( k \) and then taking its floor value. Let \( Y \) be the integer result of the scaling operation, using Axiom 1, we have
\[
Y = \left\lfloor \frac{X}{k} \right\rfloor
\]  
(14)

where \( \left\lfloor \cdot \right\rfloor \) is the least integer function.

\( k \) is chosen to be equal to \( m_2 = 2^n \)(The choice of \( k \) is crucial in ensuring that the truncation error is negligible and this will be discussed shortly), which produces
\[
\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor m_3 \left\lfloor M_1^{-1} \right\rfloor m_1 x_1 + \frac{m_1m_3}{m_2} \left\lfloor M_2^{-1} \right\rfloor m_2 x_2 + m_1 \left\lfloor M_3^{-1} \right\rfloor m_3 x_3 \right\rfloor
\]  
(15)

Using (15), the scaled integer can be computed directly from the RNS representation of \( X \).

It is desired to have the output of the RNS scaler to be generated directly in the residue form so that it can be directly processed by the arithmetic unit in each residue channel. The residue digit in each channel can be computed by taking the corresponding modulo operation on both sides of (15) as follows:
\[
\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor m_3 \left\lfloor M_1^{-1} \right\rfloor m_1 x_1 + \frac{m_1m_3}{m_2} \left\lfloor M_2^{-1} \right\rfloor m_2 x_2 + m_1 \left\lfloor M_3^{-1} \right\rfloor m_3 x_3 \right\rfloor
\]  
(16)

where \( i = 1, 2 \) and \( 3 \).

For \( m_1 \) and \( m_3 \) channels, according to Theorem 1, (16) can be simplified to
\[
\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor m_1 \left\lfloor M_1^{-1} \right\rfloor m_1 x_1 + \frac{m_1m_3}{m_2} \left\lfloor M_2^{-1} \right\rfloor m_2 x_2 + m_1 \left\lfloor M_3^{-1} \right\rfloor m_3 x_3 \right\rfloor
\]  
(17)

where \( i = 1, 3 \).

Each independently scaled residue of (17) can be further reduced to
\[
\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor m_3 \left\lfloor M_1^{-1} \right\rfloor m_1 x_1 + \frac{m_1m_3}{m_2} \left\lfloor M_2^{-1} \right\rfloor m_2 x_2 \right\rfloor
\]  
(18)
\[ \left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor \frac{m_1 m_3}{m_2} M^{-1}_{m_2} \left( x_1 + m_1 M^{-1}_{m_1} x_2 \right) \right\rfloor_{m_2} \]  
(19)

For modulus \( m_2 \),
\[ \left\lfloor \frac{X}{m_2} \right\rfloor = m_1 m_3 \left( x_1 + m_1 m_3 x_2 \right) \left( M^{-1}_{m_2} \right) \left( m_2 \right) \]  
(20)

As modulo \( 2^n \) of a variable is the same as keeping only the \( n \) least significant bits of it, the scaled residue digit, \( y_2 \), for the modulus \( m_2 \) can be obtained by simply hardwiring the \( n \) least significant bits of the result computed from the right hand side of (20). There is no additional logic and delay cost incurred by direct hardwiring.

It is observed that \( \frac{m_1 m_3}{m_2} \left( M^{-1}_{m_2} \right) \) is a common term in (18), (19) and (20). Substituting the expressions of \( m_1, m_2, m_3 \) and \( M^{-1}_{m_2} \) for this common term, we have
\[ \left\lfloor \frac{m_1 m_3}{m_2} M^{-1}_{m_2} \right\rfloor_{m_2} = \left\lfloor \frac{2^n - 1)(2^n + 1)(2^n - 1)}{2^n} \right\rfloor \]
\[ = \left\lfloor \frac{2^{2n} - 2^{n+1} + 2^{n+1} + 2^n - 2^{n+1} + 2^n - 2^n - 1}{2^n} \right\rfloor \]
\[ \approx 2^{2n} - 2^{n+1} + 2^n - 1 \]  
(21)

The least integer function \( \lfloor \cdot \rfloor \) of the scaling operation in RNS justifies the truncation of (21). The effect of this approximation will be analyzed in Section C.

By substituting (21) into (18), (19) and (20), we have
\[ \left\lfloor \frac{X}{m_2} \right\rfloor = \left( 2^{2n} - 2^{n+1} + 2^n - 1 \right) x_1 + \left( 2^{2n} - 2^{n+1} + 2^n - 1 \right) x_2 \]
\[ \left\lfloor \frac{X}{m_2} \right\rfloor = \left( 2^{2n} - 2^{n+1} + 2^n - 1 \right) x_1 \]
\[ \left\lfloor \frac{X}{m_2} \right\rfloor = \left( 2^{2n} - 2^{n+1} + 2^n - 1 \right) x_2 \]  
(22, 23, 24)

Since \( 2^n_{m_2} = \left\lfloor \frac{X}{m_2} \right\rfloor \), as a corollary of Axioms 2 and 3, the following periodic property for modulus \( m_1 \) can be defined \([35], [36]\).

**Corollary 1:** For any non-negative integer, \( a \)
\[ 2^{2n+1}_{m_1} = \left\lfloor 2^a \right\rfloor_{m_1} \]  
(25)

By applying Axioms 2 and 3 and Corollary 1, and using \( m_1 m_3 = (2^n - 1)(2^n + 1) = 2^{2n} - 1 \), (22), (23) and (24) can be reduced eventually to the following highly simplified expressions in Theorem 3.
\[ \left\lfloor \frac{X}{m_2} \right\rfloor = \left\lfloor x_1 - x_2 \right\rfloor_{m_2} \]

**Example 1:** Consider the same integer \( X = 3316100 \) from Table 1 with the RNS representation of \( (80, 132, 29) \) for the moduli set \( \{255, 256, 257\} \). Using (11), (12) and (13), the scaled output in RNS representation is calculated in Table II. It can be verified that the RNS number \( (203, 153, 103) \) is equivalent to the integer, 12953 computed directly from \( \left\lfloor \frac{3316100}{256} \right\rfloor \). Note that the binary representation of 12953 = 0011001010011001 is a byproduct in the computation of the scaled residue \( y_2 \) before the final modulo of (12) is taken. Its least eight significant bits are 10011001 = 153.

<table>
<thead>
<tr>
<th>Table II: Computation traces of scaling ((80, 132, 29)) by ( k=256 ) in RNS ( {255, 256, 257} )</th>
</tr>
</thead>
</table>
| \[ \left\lfloor \frac{X}{80} \right\rfloor \] | \( \left\lfloor \frac{32986 \times 80 - 256 \times 132 + 32985 \times 29}{255} \right\rfloor \)
| \( = 12953 \) | \( \left\lfloor \frac{32986 \times 256 - 256 \times 132 + 32985 \times 29}{256} \right\rfloor \)
| \( \left\lfloor \frac{X}{132} \right\rfloor \) | \( \left\lfloor \frac{32986 \times 256 - 256 \times 132 + 32985 \times 29}{256} \right\rfloor \)
| \( = 132 \) | \( \left\lfloor \frac{132 + 256 \times 29}{256} \right\rfloor \)

C. **Error Analysis**

The second term of the exact scaling equation of (15) is truncated to produce the approximation
\[ \left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor \left( 2^{2n+1} + 2^n - 1 \right) x_1 + \left( 2^{2n+1} + 2^n - 1 \right) x_2 \right\rfloor_{m_1} \]  
(26)

Let \( k_1 = 2^{2n+1} + 2^n - 1 \), \( k_2 = 2^{2n+1} + 2^n - 1 \), \( k_3 = 2^{2n+1} + 2^n - 1 \), and \( p = m_1 m_3 \), (26) can be rewritten as
\[ \left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor k_1 x_1 + k_2 x_2 + k_3 x_3 \right\rfloor_{m_1} \]  
(27)

The integer resulting from the above modulo operation can be expressed using the definition of (5).
\[ \left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor k_1 x_1 + k_2 x_2 + k_3 x_3 - \varepsilon p \right\rfloor \]  
(28)

where \( \varepsilon \) is a non-negative integer.

Without truncation, the exact equation, (15), is given by
\[ \left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor (2^{2n+1} + 2^{n-1})x_1 + (2^{2n} - 2^{n+1} + 2^n - 1 + \frac{1}{2^n})x_2 \right\rfloor \]
\[ = k_1x_1 + \left( k_2 + \frac{1}{2^n} \right) x_2 + k_3 \left( x_3 \right) \]
\[ = k_1x_1 + k_2x_2 + k_3x_3 \quad (29) \]
\[ = k_1x_1 + k_2x_2 + k_3x_3 + c - \epsilon p \]

where \( c = \frac{x_3}{2^n} \).

Since \( \left\lfloor k_1x_1 + k_2x_2 + k_3x_3 \right\rfloor_p = k_1x_1 + k_2x_2 + k_3x_3 - \epsilon p \) is an integer, based on the definition of the least integer function, (28) and (29) can be rewritten as:
\[ \left\lfloor \frac{X}{k} \right\rfloor = k_1x_1 + k_2x_2 + k_3x_3 - \epsilon p \quad (30) \]
\[ \left\lfloor \frac{X}{k} \right\rfloor = k_1x_1 + k_2x_2 + k_3x_3 - \epsilon p + \left\lfloor c \right\rfloor \quad (31) \]

Since \( 0 \leq x_2 < 2^n \), \( 0 \leq c < 1 \) and \( \left\lfloor c \right\rfloor = 0 \). Therefore,
\[ \left\lfloor \frac{X}{k} \right\rfloor = k_1x_1 + k_2x_2 + k_3x_3 - \epsilon p = \frac{X}{k} \quad (32) \]

This proves that the proposed RNS scaling algorithm is exact and does not introduce any scaling error.

IV. HARDWARE IMPLEMENTATION

From (11), since \( \left\lfloor -x_1 \right\rfloor_{2^n} = 2^n - 1 - x_1 \), the generation of the scaled RNS residue, \( y_1 \), requires a two-operand modulo \( 2^n - 1 \) adder, which can be implemented using a Ling parallel-prefix modulo \( 2^n - 1 \) adder [37].

The generation of the scaled RNS residue, \( y_2 \), by (12) needs some attention. The following two extensively used special properties of modulo \( 2^n - 1 \) arithmetic, proposed by Szabo and Tanaka [10] are exploited here to simplify the implementation.

Property 1: Multiplying an \( n \)-bit binary number \( x \) by \( r \) power of 2 in modulo \( 2^n - 1 \) is equivalent to a circular left shift operation,
\[ \left\lfloor 2^r x \right\rfloor_{2^n} = \text{CLS}_n(x, r). \quad (33) \]
where \( \text{CLS}_n(x, r) \) denotes a circular shift of the \( n \)-bit binary number \( x \) by \( r \) bits to the left.

Property 2: \[ \left\lfloor -2^r x \right\rfloor_{2^n} = \left\lfloor 2^r (2^n - 1 - x) \right\rfloor_{2^n} = \left\lfloor x \right\rfloor_{2^n} = \text{CLS}_n(x, r) \quad (34) \]
where \( \overline{x} \) is the 1's complement of the \( n \)-bit binary number \( x \).

Let the \( j \)-th bit of a residue \( x_j \) be represented as \( x_{j,r} \). The three residues to be scaled can then be represented in their binary forms as follows: \( x_1 = \left( x_{1,n-1}x_{1,n-2} \cdots x_{1,0} \right)_2 \), \( x_2 = \left( x_{2,n-1}x_{2,n-2} \cdots x_{2,0} \right)_2 \) and \( x_3 = \left( x_{3,n-1}x_{3,n-2} \cdots x_{3,0} \right)_2 \).

By applying Properties 1 and 2 to each term of (12), we have
\[ P_1 = 2^{2n-1} x_3 \left\lfloor \frac{x_2}{2^n} \right\rfloor_{2^n} = \text{CLS}_{2n}(x_1, 2n-1) \]
\[ = \left( x_{1,0} \overline{00} \cdots x_{n,1-n}x_{n-1} \cdots x_{1,1} \right)_2 \quad (35) \]
\[ P_2 = 2^{n-1} x_1 \left\lfloor \frac{x_2}{2^n} \right\rfloor_{2^n} = \text{CLS}_{2n}(x_1, n-1) \]
\[ = 0x_{n,0}x_{n,1-n} \cdots x_{n,0}0000 \quad (36) \]
\[ P_3 = -2^n x_2 \left\lfloor \frac{x_2}{2^n} \right\rfloor_{2^n} = \text{CLS}_{2n}(x_2, n) \]
\[ = \left( x_{n,0}x_{n,1-n}x_{n-1} \cdots x_{n,0} \right)_2 \quad (37) \]
\[ P_4 = 2^{2n-1} x_3 \left\lfloor \frac{x_2}{2^n} \right\rfloor_{2^n} = \text{CLS}_{2n}(x_1, 2n-1) \]
\[ = \left( x_{3,0} \overline{00} \cdots x_{n,1-n}x_{n-1} \cdots x_{3,1} \right)_2 \quad (38) \]
\[ P_5 = 2^{n-1} x_1 \left\lfloor \frac{x_2}{2^n} \right\rfloor_{2^n} = \text{CLS}_{2n}(x_1, n-1) \]
\[ = x_{n,0}x_{n,1-n} \cdots x_{n,0}0000 \quad (39) \]
\[ P_6 = -x_2 \left\lfloor \frac{x_2}{2^n} \right\rfloor_{2^n} = 011 \cdots x_{n,0}x_{n-1} \cdots x_{1,0} \quad (40) \]

The sum of the binary vectors \( P_1 + P_2 \) can be expressed as a single term \( Q_1 = \left( x_{1,0}x_{1,1-n}x_{1,n-2} \cdots x_{1,0}x_{1,n-1}x_{1,n-2} \cdots x_{1,1} \right)_2 \). The remaining four addends can be reduced to three binary vectors by reordering the bits of different addends at the same position followed by logic minimization. As shown in Fig. 3(a), the 1-string in \( P_3 \) is first swapped with \( \overline{x}_{3,0} \overline{x}_{3,1-n} \cdots \overline{x}_{3,0} \) in \( P_5 \). Then, the 0-string in \( P_5 \) is swapped with the corresponding 1-string that has just been transferred to \( P_6 \). After that, \( x_{n,0} \) in \( P_4 \) is swapped with \( x_{n,0} \) in \( P_5 \). Finally, the 0-string in \( P_4 \), \( x_{n,0}x_{n,1-n}x_{n-2} \cdots x_{1,0} \) in \( P_3 \) and the 1-string in \( P_6 \) are interchanged. The strings to be swapped are boxed and annotated with its order of execution in Fig. 3(a). The vectors after bit reordering are shown in Fig. 3(b). The last two vectors (in the dotted box) of Fig. 3(b) can be merged into a single constant vector of \( (11 \cdots 1)_2 \) when \( x_{n,0} = 0 \) and \( (1011 \cdots 10111 \cdots 1)_2 \) when \( x_{n,0} = 1 \).
becomes an all-zero string and \( Q_4 = "(01\ldots101\ldots00)_2 \). The
three nonzero binary vectors, \( Q_1, Q_2 \) and \( Q_4 \), can be summed
using a 2\( n \)-bit carry save adder (CSA) with end-around carry
(EAC). On the other hand, when \( x_{3,n} = 0 \), \( Q_1 \) is an all-one string.
If \( Q_1, Q_2 \) and \( Q_4 \) are summed using a 2\( n \)-bit CSA with EAC, the
EAC from the CSA will propagate all the way to the carry out,
and wrap around to the LSB. Hence, no additional 2\( n \)-bit CSA
will be generated. The \( Q \) will be implemented by \( x_{3,n} + x_{3,i} \), where
\( i = 0, 1, \ldots, n-1 \) and \( '+' \) denotes logical OR
operation. Consequently, the four binary vectors, \( Q_1 \) to \( Q_4 \) can be
added using only one 2\( n \)-bit modulo 2\( 2^n - 1 \) CSA with EAC
and \( n \) two-input OR gates, followed by a parallel-prefix modulo
2\( 2^n - 1 \) adder. The four input vectors are formed by routing the
definitions from the inputs without using any hardware
logic. It is important to note that the maximum delay of the
CSA is kept constant at one FA delay regardless of the
value of \( n \). The residue \( y_3 \) of the scaled integer output, \( Y \) for
the modulus \( m_2 \) can be taken directly from the least significant \( n \)
bits of the 2\( 2^n - 1 \) adder.

\[
P_1 : \bar{x}_{1,2n+1}, \quad \bar{x}_{1,2n-2} \ldots \bar{x}_{2,1}, \quad \bar{x}_{3,0} \quad 1 \quad 1 \quad \ldots \quad 1 \quad 1
\]

\[
P_2 : x_{3,0} \quad 0 \quad \ldots \quad 0 \quad 0 \quad x_{3,3n} \quad x_{3,3n-1} \ldots \bar{x}_{3,1}, \quad \bar{x}_{3,0} \quad 0 \quad \ldots \quad 0 \quad 0
\]

\[
P_3 : x_{3,0} \quad \bar{x}_{3,3n-1} \quad \bar{x}_{3,3n-2} \ldots \bar{x}_{3,1}, \quad \bar{x}_{3,0} \quad 0 \quad \ldots \quad 0 \quad 0
\]

\[
P_5 : 1 \quad 1 \quad \ldots \quad 1 \quad 1
\]

Figure 3: Reduction of addends.

To produce \( y_3 \) from (13), the following two axioms [32] are
needed.

**Axiom 4:**

\[ 2^{2^i} a \big|_{2^i+1} = 2^i a + 2^{2^i} \big|_{2^i+1} \]

**Axiom 5:**

\[ 2^{2^i} a \big|_{2^i+1} = [a]_{2^i+1} \]

where \( a \in \{0,1\} \) and \( i = 0, 1, \ldots, n-1 \).

By applying Axioms 4 and 5 to the second term of (13), the
variable bits of this term are wrapped to the \( n \) least significant
bit positions and its expression becomes:

\[ 2^i x_j \big|_{2^i+1} = \sum_{j=0}^{2^i} 2^{2^i} x_j \big|_{2^i+1} = x_{j,n} + \sum_{j=0}^{2^i-1} 2^{2^i} x_j \big|_{2^i+1}, \quad (41) \]

The addition of \( x_2 \) to (41) can be reduced by a CSA tree
before a diminished-one mod 2\( ^n + 1 \) adder [28] is used to
produce the final output. A one needs to be subtracted from
each of the sum and carry outputs of the CSA tree to convert
them into diminished-one representation before the
diminished-one adder, and the output of the diminished-one
adder needs to be incremented by one to obtain the actual
output. This aggregate offset of \(-1\) can be added upfront in the
CSA tree. According to Axiom 4, the carry generated out of the
most significant bit of each \( n \)-bit CSA will be complemented
and added to the least significant bit of the next level of CSA
together with a constant addition of 2\( n \). By assuming a
three-level CSA tree, an offset of 3\( \times 2^n \) will be generated.
Together with the compensation for diminished one addition
and the last term of (41), the total correction constant, \( C \) to be
added in the CSA tree is given by:

\[
C = \left[ \sum_{i=0}^{n} 2^i - 1 + 3 \times 2^n \right]_{2^n+1} = [2^{2n} - 2^n - 1 + 3 \times 2^n]_{2^n+1}
\]

\[
= [1 - (1 - 1 + 3 \times (1)]_{2^n+1} = -[2^{2^n}]_{2^n+1}
\]

\[ (42) \]

The addends to be summed in the CSA tree are shown in Fig.
4(a). The first \( n \)-bit CSA adds only the last two addends to
produce the results of Fig. 4(b), with the complementary
end-around carry (CEAC) printed bold. Since \( 1 + a = 2a + \bar{a} \)
for any binary variable \( a \), no new signal variable is introduced. The
result can be obtained by direct hardwiring without the need for
the CSA. The second \( n \)-bit CSA adds the last three addends of
Fig. 4(b) to produce the results of Fig. 4(c), with the CEAC
printed bold. Based on the property of \( 1 + a = 2a + \bar{a} \), only two
modified full adders (MFAs), instead of a complete \( n \)-bit CSA,
are required to produce the unknown variables, \( s_0, c_0, s_1 \) and \( c_1 \).
Finally, an \( n \)-bit CSA is required to reduce Fig. 4(c) to two
addends to be fed into the final diminished-one mod 2\( ^n + 1 \)
adder. Although we started with a three-level CSA addition,
most intermediate results can be directly hardwired and only
one level of \( n \)-bit CSA with CEAC and two MFAs are needed
eventually, which are highlighted by the shaded boxes of Fig. 4.

![Figure 4: Simplification of the CSA tree for the computation of \( y_3 \)](image-url)
The complete RNS scaling architecture is shown in Fig. 5. Since the inner modulo $2^n - 1$ operation in (12) has to be computed any way to obtain the scaled output, $y_2$, the scaled integer, $Y$ is also available as a byproduct of this computation. Thus, without introducing any hardware overhead, the scaled output is available in both RNS and binary forms. This flexibility to manipulate the scaled output in two domains has advantage in interfacing the scaled output to computational units in either domain without the need for the expensive RNS-to-binary converter. For example, the bottleneck reverse converter in the last stage of Fig. 1(a) can be eliminated to increase the throughput of the RNS IPSP.

![Figure 5: Architecture of the proposed RNS scaler.](image)

**Example 2:** Consider the RNS representation of $X = (80, 132, 29)$ for the moduli set $(255, 256, 257)$ from Example 1. $x_1 = 01010000_2$, $x_2 = 10000100_2$, and $x_3 = 000011101_2$. $Q_1 = 0001010000_2$, $Q_2 = 01110111111000102$, $Q_3 = 1000111010011110_2$ and $Q_4 = 1111111111111_2$. The modulo $2^n - 1$ addition of $x_1$ and $X$, carry save addition with end-around carry of the four $2n$-bit binary vectors, $Q_1$ to $Q_4$, and diminished-one modulo $2^n + 1$ addition of $x_2$ and $x_3$ are illustrated in Fig. 6. From Fig. 6, $Y = 00110010101100101_2 = 12953$, $y_1 = 110011011_2 = 203$, $y_2 = 100110101_2 = 153$ and $y_3 = 0110111_2 = 103$ tally with the correct outputs from Table II.

In the final part of this section, we show that the final modulo $2^{2n} - 1$ adder in $m_1$ channel can be replaced by a much simpler modulo $2^n$ adder if only the residue output $y_2$ is needed.

From [37],

$$|A + B|_{2^n - 1} = \begin{cases} |A + B|_{2^n} + 1 & \text{if } A + B \geq 2^n \\ |A + B|_{2^n} & \text{Otherwise} \end{cases} \quad (43)$$

It follows that

$$|A + B|_{2^{2n} - 1} = \begin{cases} |A + B|_{2^n} + 1 & \text{if } A + B \geq 2^n \\ |A + B|_{2^n} & \text{Otherwise} \end{cases} \quad (44)$$

Using Theorem 1,

$$|A + B|_{2^{2n} - 1} = \begin{cases} |A + B|_{2^n} + 1 & \text{if } A + B \geq 2^n \\ |A + B|_{2^n} & \text{Otherwise} \end{cases} \quad (45)$$

Only a few additional logic gates are needed to compute the carry-in to the modulo $2^n$ adder based on the Ling’s carry definition [37].

![Figure 6: Calculation of scaled RNS output, $y_1$, $y_2$ and $y_3$ for Example 2.](image)

**V. Evaluation and Comparison**

In this section, the total delay and hardware area of the proposed design are modeled and simulated in order to analyze its hardware implementation cost and complexity. In order to benchmark our proposed RNS scaler architecture against the FA based scaler architecture [19] and other RNS scaler architectures compared in [19], the unit-gate model used in [28] is adopted for our analysis. In this analytical model, a two-input monotonic gate, such as AND or NAND gate, is said to have one unit of area and one unit of delay. An XOR gate is deemed to consume two units of area and have two units of delay. The area and delay of an inverter is a negligible fraction of a unit and hence it is assumed to have zero unit of area and delay. Based on the unit-gate model, a FA has seven units of area and 4 units of delay.

Based on the architecture (See Fig. 5) described in Section IV, the area and delay for each residue channel can be independently evaluated by studying the area and time complexity of the logic gate implementation of the carry save adder tree, if any, and the carry propagating adder (CPA).

Based on Fig. 6 of [37], the area and delay for the Ling modulo $2^n - 1$ adder are estimated to be $3n \left[ log_2 n - 1 \right] + 12n$ and $2 \left[ log_2 n - 1 \right] + 3$ units, respectively. From Table 2 of [28], the area and time complexity for the diminished-one mod $2^n + 1$ adder are $4.5n \left[ log_2 n \right] + 0.5n + 6$ and $2 \left[ log_2 n \right] + 3$ units, respectively. For the $m_1$ channel, we need $n$ inverters for the one-complement of an operand before the CPA. A $2n$-bit CSA with EAC and $n$ OR gates are required in the $m_2$ channel. $2n$ FAs are required to implement the CSA. Finally, from Fig. 4, two MFAs and $n$ FAs are required to implement the CSA with CEAC for the $m_3$ channel. The decomposition of area and
delay for each channel is tabulated in Tables III and IV, respectively.

**TABLE III: ESTIMATION OF UNIT-GATE AREA OF PROPOSED DESIGN**

<table>
<thead>
<tr>
<th>Modulus</th>
<th>$A_{\text{FA}}$</th>
<th>$A_{\text{other}}$</th>
<th>$A_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1$</td>
<td>$3n \log_2 n - 1 + 12n$</td>
<td>$= 0$</td>
<td>$3n \log_2 n - 1 + 12n$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$6n \log_2 n + 24n$</td>
<td>$15n$</td>
<td>$6n \log_2 n + 39n$</td>
</tr>
<tr>
<td>$m_3$</td>
<td>$4.5n \log_2 n + 0.5n + 6$</td>
<td>$7n + 6$</td>
<td>$4.5n \log_2 n + 7.5n + 12$</td>
</tr>
</tbody>
</table>

**TABLE IV: ESTIMATION OF UNIT-GATE DELAY OF PROPOSED DESIGN**

<table>
<thead>
<tr>
<th>Modulus</th>
<th>$D_{\text{FA}}$</th>
<th>$D_{\text{other}}$</th>
<th>$D_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1$</td>
<td>$2 \log_2 n - 1 + 3$</td>
<td>$= 0$</td>
<td>$2 \log_2 n - 1 + 3$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$2 \log_2 n + 3$</td>
<td>$5$</td>
<td>$2 \log_2 n + 8$</td>
</tr>
<tr>
<td>$m_3$</td>
<td>$2 \log_2 n + 3$</td>
<td>$6$</td>
<td>$2 \log_2 n + 9$</td>
</tr>
</tbody>
</table>

It is worth noting here that the delays of the non-CPA logic in all three modulus channels are independent of the word length, $n$. In addition, the delay of the carry propagating addition is only logarithmically dependent on the word length $n$. This means that the rate of increase in delay actually slows down as $n$ grows larger. In fact, the speed is merely dependent on the performance of the modulo $2^n + 1$ and $2^n - 1$ adders. In other words, the proposed scaling algorithm has successfully circumvented the complexity of RNS scaling problem for $\{2^n - 1, 2^n + 1\}$ by reducing it into an operation as simple as addition or constant multiplication in the same RNS.

We also benchmark the area and time complexity of our proposed RNS scaler against other designs reported in the RNS scaling architecture [19], including [19] itself. For ease of reference, the algorithms and scaling errors of these designs are listed in Table V. All the implementations are based on ROMs except for the proposed scaler, Dasygenis08 [19] and Garcia99 [17]. The latter is based partly on ROM-based arithmetic and partly on FAs. Since Garcia99 has significantly improved performance over Garcia99B from the same authors, Garcia99B in [19] is not listed here for comparison. From Table V, it is noted that only the proposed design and the two-lookup cycle design have no scaling error. The other scaling methods introduce some non-zero scaling error. Even though the error may be small, it could escalate to an unacceptable magnitude after several iterations of computation as shown in Fig. 1. As discussed in Section II, Dasygenis08 [19] is not able to produce the correct scaled output and it is not meaningful to analyze its scaling error.

**TABLE V: ALGORITHMS AND SCALING ERRORS OF SCALERS IN COMPARISON**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Algorithm</th>
<th>Scaling error</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>FA based</td>
<td>0</td>
</tr>
<tr>
<td>Dasygenis08 [19]</td>
<td>FA based</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Garcia99 [17]</td>
<td>Two lookup cycle</td>
<td>0</td>
</tr>
<tr>
<td>Ulman93 [22]</td>
<td>CRT based</td>
<td>1</td>
</tr>
<tr>
<td>Shenoy89 [14]</td>
<td>CRT decomposition</td>
<td>1.5</td>
</tr>
<tr>
<td>Griffin88 [21]</td>
<td>L-CRT</td>
<td>N</td>
</tr>
</tbody>
</table>

For consistency, we adopt the same method of evaluation as [19] so that the hardware area and time complexity of the contending designs can be excerpted directly from Table IV of [19] for comparison. Since the hardware complexity is expressed in terms of the number of transistors for all designs, the transistor count of our proposed design can be estimated from its unit-gate area by reasonably and conservatively assumed that a two-input monotonic gate can be implemented with six transistors using a classical CMOS implementation. In addition, we consider an inverter as equivalent to two transistors. The conversion of the unit gate area from Table III to the equivalent number of transistors for our proposed design is shown in Table VI.

The same unit gate delay model is used in [19] for the comparison of time complexity except that a classic FA is assumed to have only 2 units of delay in [19], which is equivalent to that of an XOR gate. This estimate is optimistic because the delay of a classic 28T FA implemented in static CMOS technology [39]-[41] is at least 3 units. In other words, the total delay of Dasygenis08 [19] has been underestimated.

**TABLE VI: ESTIMATED NUMBER OF TRANSISTOR OF PROPOSED DESIGN**

<table>
<thead>
<tr>
<th>Modulus</th>
<th>$A_{\text{FA}}$</th>
<th>$A_{\text{other}}$</th>
<th>$A_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1$</td>
<td>$18n \log_2 n - 1 + 72n$</td>
<td>$2n$</td>
<td>$18n \log_2 n - 1 + 74n$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$36n \log_2 n + 144n$</td>
<td>$90n$</td>
<td>$36n \log_2 n + 234n$</td>
</tr>
<tr>
<td>$m_3$</td>
<td>$27n \log_2 n + 3n + 36$</td>
<td>$42n + 36$</td>
<td>$27n \log_2 n + 45n + 72$</td>
</tr>
</tbody>
</table>

As the use of RNS scaler is to ensure that the computed result represents the real value instead of its residue modulo $M$, it is fairer to compare different RNS scalers based on the same dynamic range. As this is not always possible due to the odd values of moduli selected by [19], the value of $n$ for our proposed design has been selected such that its dynamic range is at least several time larger than the dynamic range of other designs so that the comparison is always in favor of the contenders. Tables VII and VIII show the comparisons of the transistor counts and the delay, respectively, of six different RNS scaling architectures for four different dynamic ranges.

**TABLE VII: COMPARISON OF ESTIMATED NUMBER OF TRANSISTORS (VALUE IN PARENTHESIS INDICATES THE PERCENTAGE AREA REDUCTION OF PROPOSED DESIGN OVER ITS CONTENDER)**

<table>
<thead>
<tr>
<th>Proposed ($m_1$, $m_2$, $m_3$)</th>
<th>[19] ($m_1$, $m_2$)</th>
<th>Number of transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{This}$</td>
<td>$[19]$</td>
<td>$[17]$</td>
</tr>
<tr>
<td>(7,8,9)</td>
<td>(5,17)</td>
<td>1563</td>
</tr>
<tr>
<td>(15,16,17)</td>
<td>(10,12,13)</td>
<td>2060</td>
</tr>
<tr>
<td>(31,32,33)</td>
<td>(18,19,20)</td>
<td>2962</td>
</tr>
<tr>
<td>(255,256, 257)</td>
<td>(25,26, 27)</td>
<td>10961</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposed ($m_1$, $m_2$, $m_3$)</th>
<th>[19] ($m_1$, $m_2$)</th>
<th>Number of transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,8,9)</td>
<td>(5,17)</td>
<td>1563</td>
</tr>
<tr>
<td>(15,16,17)</td>
<td>(10,12,13)</td>
<td>2060</td>
</tr>
<tr>
<td>(31,32,33)</td>
<td>(18,19,20)</td>
<td>2962</td>
</tr>
<tr>
<td>(255,256, 257)</td>
<td>(25,26, 27)</td>
<td>10961</td>
</tr>
</tbody>
</table>

It is obvious from Tables VII and VIII that the proposed design is significantly faster and has much lower hardware complexity than all the other designs. The proposed design reduces 79.09%, 80.61%, 75.19% and 60.67% of the transistor counts from the design of [19] for an extended DR of 504 ($= 6$..)
Although Tables VII and VIII provide a good insight on the relative competitiveness of various designs based on the same method of evaluation as [19], the moduli sets used are different from ours. For a more accurate evaluation and fairer comparison, the fastest ROM-based design [17] from Table VIII has been selected for implementation using the same three moduli set \( \{2^n - 1, 2^n, 2^n + 1\} \) and scaling factor \( 2^n \). In addition, another fast and precise base-extension ROM-based design [15] and the current-art adder-based design [24] that were not compared in [19] are also implemented using the same moduli set and scaling factor. According to the original algorithms, the scaling constant of [15] can be of any subset of the moduli set. However, the use of a product of lowest order moduli as scaling constant in the example of [15] may not necessarily lead to more efficient hardware implementation. It is worth noting that the [15] and [24] have zero scaling error. Unit gate area and delay of these designs are also analyzed. For this theoretical analysis, the design [24] without the sign number handling is evaluated against our design without the scaled integer output.

Since the designs, [15] and [17], are implemented using ROMs, a ROM model based on [16] is used to estimate the number of transistors, \( A_{ROM} \) and the unit gate delay, \( T_{ROM} \). For a \( 2^p \times p \) ROM [16], \( A_{ROM(2^p \times p)} = 2^{\lceil \log_2 p \rceil + \lceil \log_2 n \rceil + 1} + 2^{\lceil \log_2 p \rceil} \cdot 2^{\lceil \log_2 n \rceil} \) and \( T_{ROM(2^p \times p)} = (1 + \log_2 n + \lceil \log_2 n \rceil) \cdot T_{NAND} \). It is stated in [17] that, for three moduli set with one of the modulus being the scaling constant, the design requires three double-input ROMs. Therefore, the total number of transistors of the design [17] is estimated to be 
\[
A_{ROM(2^n \times n)} + A_{ROM(2^n \times (n+1))} + A_{ROM(2^n \times n)},
\]
where \( n \) is the bit width of the moduli. Also, its total unit gate delay is approximately \( (T_{ROM(2^n \times n)} + 2n) \) units. Based on Fig. 2 of [15], additional block of hardware should be included in each of \( m_1 \) and \( m_2 \) channels of [15] to perform the base extension operation of \( X_i \), for \( i = 1 \) and 3. In this special case where \( k = 2^n \) and \( m_1 > m_1 \), only one ROM is required in the \( m_1 \) channel. As a result, the total hardware of the design [15] is estimated to be \( A_{ROM(2^n \times n)} + A_{ROM(2^n \times (n+1))} + A_{ROM(2^n \times n)} \) and its total unit gate delay is \( T_{ROM(2^n \times n)} + T_{ROM(2^n \times (n+1))} + T_{ROM(2^n \times n)} \) units.}

### Table IX: Comparison of estimated total number of transistors (A) and unit gate delay (T) for the same three moduli sets

<table>
<thead>
<tr>
<th>( m_1, m_2, m_3 )</th>
<th>This</th>
<th>[24]</th>
<th>[17]</th>
<th>[15]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>T</td>
<td>A</td>
<td>T</td>
</tr>
<tr>
<td>( 31, 32, 33 )</td>
<td>2095</td>
<td>17</td>
<td>2077</td>
<td>21</td>
</tr>
<tr>
<td>( 28, 64, 65 )</td>
<td>2478</td>
<td>17</td>
<td>2478</td>
<td>21</td>
</tr>
<tr>
<td>( 127, 128, 129 )</td>
<td>2861</td>
<td>21</td>
<td>2873</td>
<td>21</td>
</tr>
<tr>
<td>( 255, 256, 257 )</td>
<td>3244</td>
<td>17</td>
<td>3280</td>
<td>31</td>
</tr>
</tbody>
</table>

All these full-feature designs (with sign handling as reported in [24] and with both scaled residue and integer outputs for ours) are coded using Verilog HDL and functionally verified using ModelSim. The designs are synthesized and mapped to STM 90nm standard cell library using Synopsys Design Compiler. Each design is optimized for speed to obtain their minimum achievable delay. The synthesized area in \( \mu \text{m}^2 \) and critical path delay in \( \text{nS} \) of these \( \{2^n - 1, 2^n, 2^n + 1\} \) RNS scalers for \( n = 5, 7 \) and 8 are shown in Table IX. Due to the memory quota, the two LUT-based scalers are implemented only for \( n = 5 \) and 7.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area</th>
<th>Delay</th>
<th>Area</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>3168.8</td>
<td>0.87</td>
<td>4500.2</td>
<td>0.88</td>
</tr>
<tr>
<td>[24]</td>
<td>4166.5</td>
<td>1.47</td>
<td>6786.5</td>
<td>1.58</td>
</tr>
<tr>
<td>[17]</td>
<td>12315.0</td>
<td>0.98</td>
<td>13782.6</td>
<td>1.67</td>
</tr>
<tr>
<td>[15]</td>
<td>10147.3</td>
<td>1.22</td>
<td>15645.2</td>
<td>1.87</td>
</tr>
</tbody>
</table>
From Table X, it is obvious that our proposed RNS scaler consumes the least area for all values of \( n \). Comparing with the next most area efficient design [24], our proposed design consumes 23.9\%, 32.4\% and 35.1\% less area for \( n = 5, 7 \) and 8, respectively. Besides, our proposed design is faster than it by 40.8\%, 44.3\% and 44.0\% for \( n = 5, 7 \) and 8, respectively. The significant performance improvement over [24] is attributable to the different approaches to the formulation of scaling problem although both scaling algorithms are designed based on adders. From the synthesis results, it is observed that the critical path delay of our proposed design remains relatively constant for three different values of \( n \). This is consistent with the inference from the unit gate delay estimation in Table IV. Comparing with the fastest design [17] from Table VIII, our design is faster than it by 11.2\% and 47.3\% for \( n = 5 \) and 7 respectively.

The power consumption of the above designs were also simulated using PrimeTime PX with the same technology library. The total power consumption as well as the leakage power for \( n = 5, 7 \) and 8 are listed in Table XI. The results show that our proposed design dissipates the least total power as well as leakage power for all values of \( n \). It consumes 39.3\%, 50.2\% and 43.4\% less total power and 37.0\%, 44.1\%, 43.1\% less leakage power than the next most power-efficient design [24] for \( n = 5, 7 \) and 8, respectively.

<table>
<thead>
<tr>
<th>Design</th>
<th>( n = 5 )</th>
<th>( n = 7 )</th>
<th>( n = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Leakage</td>
<td>Total</td>
</tr>
<tr>
<td>This</td>
<td>3.796</td>
<td>0.670 (17.7%)</td>
<td>4.955</td>
</tr>
<tr>
<td>[24]</td>
<td>6.258</td>
<td>1.063 (17.0%)</td>
<td>9.948</td>
</tr>
<tr>
<td>[17]</td>
<td>13.761</td>
<td>3.140 (22.8%)</td>
<td>16.607</td>
</tr>
<tr>
<td>[15]</td>
<td>21.083</td>
<td>2.473 (11.7%)</td>
<td>19.313</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

In this paper, a novel area-efficient, high-speed and precise RNS scaler for the three-moduli set RNS \([2^n-1, 2^n, 2^n+1]\) is proposed. The new scaling algorithm is formulated based on the Chinese New Remainder Theorem and it uniquely exploits the number theoretic properties of this moduli set and the scaling factor of \( 2^n \) to overcome the complexity associated with the hardware implementation of inter-modulo operation. The proposed RNS scaler has an area complexity of \( O(n \log_2 n) \) and a time complexity of \( O(2^n \log_2 n) \). The integer scaled output in normal binary number representation is also generated as a byproduct of this new formulation. Hence, the expensive residue-to-binary converter can be saved if the result after scaling is also required by a normal binary number system. Our synthesis results based on 90nm standard-cell based implementation show that the proposed RNS scaler is approximately 30.5\% more area efficient and 43.0\% faster than the state-of-the-art adder-based scaler and at least 66\% more area efficient and 11.2\% faster than the fastest ROM-based scaler design. On average, the total power consumption and leakage power of our proposed design are respectively 44.3\% and 41.4\% smaller than the most power-efficient adder-based scaler design.

REFERENCES


