A Novel Modulo $2^n - 2^k - 1$ Adder for Residue Number System

Shang Ma, Jian-Hao Hu, Member, IEEE, and Chen-Hao Wang

Abstract—Modular adder is one of the key components for the application of residue number system (RNS). Moduli set with the form of $2^n - 2^k - 1$ ($1 \leq k \leq n - 2$) can offer excellent balance among the RNS channels for multi-channels RNS processing. In this paper, a novel algorithm and its VLSI implementation structure are proposed for modulo $2^n - 2^k - 1$ adder. In the proposed algorithm, parallel prefix operation and carry correction techniques are adopted to eliminate the re-computation of carries. Any existing parallel prefix structure can be used in the proposed structure. Thus, we can get flexible tradeoff between area and delay with the proposed structure. Compared with same type modular adder with traditional structures, the proposed modulo $2^n - 2^k - 1$ adder offers better performance in delay and area.

Index Terms— Carry correction, modular adder, parallel prefix, residue number system (RNS), VLSI.

I. INTRODUCTION

RESIDUE number system (RNS) is an ancient numerical representation system. It is recorded in one of Chinese arithmetical masterpieces, the Sun Tzu Suan Jing, in the 4th century and transferred to European known as Chinese Remainder Theorem (CRT) in the 12th century. RNS is a non-weighted numerical representation system and has carry-free property in multiplication and addition operations. In recent years, it has been received intensive study in the very large scale integration circuits (VLSI) design for digital signal processing (DSP) systems with high speed and low power consumption [1]–[4]. Modular adder is one of the key modules for RNS-based DSP systems.

For integers $A$ and $H$ with $n$-bit width, the modular addition can be performed by (1) if $A$ and $H$ is less than the modulus $m$

$$C = (A + H)_m = \begin{cases} A + H, & A + B + T < 2^n \\ (A + B + T)_2, & A + B + T \geq 2^n \end{cases}$$  

(1)

In (1), $T = 2^n - m$, which is referred as correction [5]–[8]. In the general modular adder design, the two values, $A + H$ and $A + B + T$, should be computed firstly. Then, one of them is selected as the final output. According to the form of the modulus, modular adders can be classified into two types: the general modular adder and the special modular adder.

For the general modular adder, Bayoumi proposed a scheme for arbitrary modulus by using two cascaded binary adders [5]. However, the delay is the sum of the two binary adders. Several literatures constructed several modular adders with two parallel binary adders to calculate $A + H$ and $A + H + T$ [6], [7]. This method can achieve less delay but needs about twice area of binary adder. Dugdale proposed a method to construct a type of general modular adders with a reused binary adder [9]. The shortage of this structure is that it will use two operation cycles to perform one modular addition. The area or delay of these modular adders mentioned above is twice or more than that of binary adder. In recent studies, a few modular adders with better area and delay performance are presented. Hiasat proposed a class of modular adders in which any regular Carry Look-Ahead (CLA)—based binary adder can be used in the final stage [10]. However, it needs an extra CLA unit to get the carry-out bit of $A + B + T$ before the final CLA addition. As a result, the structure does not reduce the delay significantly. The ELMMA algorithm proposed by Patel et al. [11] uses two carry computation modules for $A + B$ and $A + B + T$ in which some carry computation units can be shared. The area reduction of this scheme is dominated by the form of $T$. In the worst case, almost two independent carry generation modules are needed. Patel et al. [12] also proposed several algorithms which can generate carries faster. A new number representation for modulo addition is proposed in [8]. However, its outputs are represented in special format. Thus, the extra area and delay are needed to perform the conversion from the special representation to binary representation or all operations should be performed in this number representation format in RNS-based systems.

On the other hand, the complexity of the special modular adder is much less than that of general modular adder, since the structure of the special modular adder can be further optimized according to the modulus. The effective modular adders for modulo $2^n - 1$ and modulo $2^n + 1$ have drawn much more attention than other kinds of modular adders [13], [14], [15] and [16] proposed an architecture for modulo $2^n + 1$ adder based on “diminished-1” number representation. [17] and [18] presented a structure for modulo $2^n + 1$ and $2^n - 1$ based on parallel prefix and carry correction, respectively. A similar architecture with $[7]$ for modulo $2^n + 3$ adder is also proposed in [19]. In [20], Patel et al. described an implementation structure for modulo $2^n - 2^n - 2 - 1$ adder based on the technique of carry offset, which is only required to obtain the carry information of $A + B + T$. In order to obtain the carries required in the modular addition, each carry of $A + B + T$ has to be modified according to the utmost carry of $A + B + T$. In this case, the redundant modules of carry computation are eliminated, but the structure of carry
computation is fixed and can only perform the special modular addition, that is, modulo $2^n - 2^{n-2} - 1$ addition.

One of the important issues is the selection of moduli sets in RNS-based application. In addition and multiplication intensive systems, residue channels are always expected as many as possible when the dynamic range is fixed, that is, the word length of individual residue can be reduced to achieve better speed performance. Meanwhile, the width of each channel is also expected as close as possible to get similar critical path delay. That is the balance between each residue channel. Moreover, the complexity of modular adder should be evaluated carefully in residue radix selection. At present, it is possible to get high performance modular adders for a few moduli radixes, such as modulo $2^n - 1$ and modulo $2^n + 1$. But these moduli radixes are not always suitable to construct multi-channel RNS with fine channel balance. For example, it is hard to construct a multi-channel moduli set with $2^n - 1$ and $2^n + 1$ to achieve co-prime and fine balance between channels. However, the modulus with the form of $2^n - 2^k - 1 \{1 \leq k \leq n - 2\}$ have the prominent advantage in constructing multi-channel moduli sets with fine balance [21]. We can find several methods for moduli set selection with this type residue. For instance, we can verify that the moduli set $\{2^n, 2^n - 1, 2^n - 2^1 - 1, \ldots, 2^n - 2^k - 1, \ldots, 2^n - 2^{n-2} - 1\}$ satisfies the co-prime requirement when $n = 3, 4, 5, 6, 8, 12$, and when $n = 7, 9, 10, 11$ by removing a few radices. Meanwhile, the channel widths of these moduli sets are all $n$ bits. Thus, the residue radix with the form of $2^n - 2^k - 1$ has great potential in moduli sets constructing with high efficient, high dynamic range, and fine balance between channels. Due to the advantages of radix $2^n - 2^k - 1$, it is essential to study its fundamental computation units, that is, modulo $2^n - 2^k - 1$ adder and modulo $2^n - 2^k - 1$ multiplier. In [21], a general architecture for modulo $2^n - 2^k - 1$ multiplier is proposed recently. A modulo $2^n - 2^{n-2} - 1$ and a modulo $2^n + 3$ adder are also proposed in [19] and [20] respectively. However, there is little discussion about the general architecture for modulo $2^n - 2^k - 1$ adder.

![Fig. 1. Prefix computation-based adder structure.](image)

In this paper, a new class of modulo $2^n - 2^k - 1$ adder based on carry correction and parallel prefix algorithm is proposed. The new modular adder can be divided into four units, the pre-processing unit, the prefix computation unit, the carry correction unit, and the sum computation unit. In the proposed scheme, the carry information of $A + B + T$ computed by prefix computation unit is modified twice to obtain the final carries required in the sum computation module. Meanwhile, any existing fast prefix structure of binary adders can be used in the proposed modular adder structure, which offers superior flexibility in design. In order to evaluate the performance of the proposed modular adder in this paper, the unit-gate model and Design Compiler (DC) of Synopsys Company are used to estimate its complexity and performance. The results show that the proposed modulo $2^n - 2^k - 1$ adder can get the best delay performance. Compared with the special modulo $2^n - 2^{n-2} - 1$ adder proposed in [20], our method offers similar delay performance but has the ability of design a class of modulo $2^n - 2^k - 1$ adder with different $k$ based on identical algorithm. Moreover, compared with ELMMA modular adder, the proposed modulo adder has better “area*delay” performance at most cases and can achieve faster operation frequency.

In the rest of the paper, the brief introduction of RNS and modular addition are presented in Section II. Section III introduces the algorithm and hardware architecture of the proposed modulo $2^n - 2^k - 1$ adder. Performance of the proposed modular adder are evaluated and compared with other modular adders in Section IV. Finally, we will conclude this paper.

II. BACKGROUND

A. RNS and Modular Addition

RNS is defined as a group of co-prime modular radixes $\{m_i, m_2, \ldots, m_N\}$, where $N > 1$, $gcd(m_i, m_j) = 1$, $i \neq j$, $i, j = 1, 2, \ldots, N$, and $gcd(m_i, m_j)$ is the greatest common divisor of $m_i$ and $m_j$. The integer $X$ in \{0, $M$\} can be represented uniquely by its residues respect to the modulus $m_i$, that is, $\{x_1, x_2, \ldots, x_N\}$, where $x_i = \{X\}_{m_i}$, $M = \prod_{i=1}^{N} m_i$, $i = 1, 2, \ldots, N$. Let $\{a_1, a_2, \ldots, a_N\}$, $\{b_1, b_2, \ldots, b_N\}$ and $\{c_1, c_2, \ldots, c_N\}$ be the RNS representation of integers $A$, $B$ and $C$ in the range of \{0, $M$\}. According to Gaussian modular algorithms, if $c_i = \{a_i \Delta b_i\}_{m_i}$, we can get $C = \{A \Delta B\}_M$, where “$\Delta$” represent addition, subtraction, and multiplication.

For integers $A$ and $B$ in the range of \{0, $m$\}, modulo $m$ addition is defined as

$$C = \{A + B\}_m = \begin{cases} A + B & A + B < m \\ A + B - m & A + B \geq m. \end{cases}$$

(2)

If $C = \{A + B\}_m$ and the bit width of the modular adder is $n$-bit, where $n = \lfloor \log_2 m \rfloor$ (that is, $n$ is the smallest integer no less than $\log_2 m$). Equation (2) can be represented as

$$C = \begin{cases} A + B & A + B + T < 2^n \\ (A + B + T)_2 & A + B + T \geq 2^n \end{cases}$$

(3)

where the correction $T = 2^n - m$ [7], [8], [20]. That is, if the carry-out bit of $A + B + T$ is “1”, the result of modular addition is the least significant bits of $A + B + T$, otherwise, the result is $A + B$. This is the basic rule in most modular adders design.

B. Prefix Parallel Addition

Parallel prefix operation is widely adopted in binary adder design. Each sum bit $s_i$ and carry bit $c_i$ can be calculated with the previous $i$ carries and inputs [22]. As shown in Fig. 1, prefix-based binary adders can be divided into three units, the pre-processing unit, the prefix computation unit, and the sum computation unit.

In the pre-processing unit, prefix computation is calculated as

$$\{g_i, p_i\} = \{a_i b_i, a_i \oplus b_i\}$$

(4)
where \( g_i \) and \( p_i \) represent the \( i^{th} \) \((i = 0, 1, \ldots, n - 1)\) carry generation bit and carry propagation bit respectively.

The prefix computation unit is used to compute the carry information used in the sum computation unit. Prefix computation can be performed by

\[
\begin{aligned}
\left( G_{i}^{G}, P_{i}^{G} \right) &= (g_i, p_i) \\
\left( G_{i}^{C}, P_{i}^{C} \right) &= \left( G_{i}^{C,i+1}, P_{i}^{C,i+1} \right) \cdot \left( G_{i}^{C,i}, P_{i}^{C,i} \right)
\end{aligned}
\]

where \( i = 0, 1, \ldots, n - 1 \), \( 0 \leq k \leq j \leq i \), \( l = 1, 2, \ldots, m \), and \( i \) represents the \( i^{th} \) stage. The smaller \( l \) means the shorter delay of the carry chain. The operator \( \text{in}(5) \) is the prefix operator and \( (G_{i}^{C}, P_{i}^{C}) \) is the prefix computation result of the \( l^{th} \) stage from the \( k^{th} \) bit to the \( i^{th} \) bit, which is also called group prefix computation. There are several well-known binary prefix addition structures, such as Sklansky (SK), Brunt-Kung (BK), Kogge-Stone (KS), Han-Carlson (HC), ELM, and so forth [22]. The prefix structures mentioned above are usually called prefix trees.

After prefix computation, carries \( c_i \) \((i = 0, 1, \ldots, n)\) for the \( i^{th} \) bit can be obtained. They can be computed as

\[
\begin{aligned}
c_{0} &= c_{\text{fin}} \\
c_{i} &= G_{i}^{C,i-1,0} + P_{i}^{C,i-1,0} c_{\text{fin}} \quad i = 1, 2, \ldots, n \\
c_{\text{out}} &= c_{n}
\end{aligned}
\]

In the sum computation unit, the carries \( c_i \) from the prefix computation unit and the partial sum \( p_i \) from the pre-processing unit are used together to compute the final sum bits \( s_i \),

\[
s_i = p_i \oplus c_i \quad i = 0, 1, \ldots, n - 1
\]

C. Unit-gate Model for Area and Delay Analysis

The unit-gate model is one of the most commonly used models to estimate the circuit complexity and performance in VLSI design. In the unit-gate model, simple two-input logic gates, such as AND, OR, NAND, and NOR, are treated as unit gates. They have the same area and delay, which are referred as \( A_u \) and \( T_u \) in this paper, respectively. For those more complicated two-input gates, such as XOR and XNOR, their area and delay are defined as \( A_{u} \) and \( T_{u} \) in our analysis, respectively. Complex logical circuits as well as multi-input gates can be implemented with 2-input unit gates, and their gate counts equal the sum of gate counts of the unit gate [22].

III. PROPOSED MODULO \( 2^n - 2^k - 1 \) ADDER

As shown in Fig. 2, the proposed modulo \( 2^n - 2^k - 1 \) adder is composed of four modules, pre-processing unit, carry generation unit, carry correction unit, and sum computation unit. In Fig. 2, different shade represents different processing units.

The proposed modular adder can be divided into two general binary adders, \( A1 \) and \( A2 \) in Fig. 2, with carry correction and sum computation module according to the characteristics of correction \( T \) for modulus \( 2^n - 2^k - 1 \). We can get the carries \( c_i^{\text{real}} \) used in the final stage through correcting the carries \( c_i^T \) of \( A + B + T \), which can be computed by any existing prefix structure with proper pre-processing. At last, we can get the final modular addition result from \( c_i^{\text{real}} \) and partial sum information. The proposed architecture shown in Fig. 2 can avoid the calculation of carries information for \( A + B + T \) and \( A + B \) separately. Thus, the area and delay in VLSI implementation can be reduced. Meanwhile, the proposed scheme offers flexible tradeoff of area and delay with different parallel prefix structures.

A. Pre-Processing Unit

The pre-processing unit is used to generate the carry generation and carry propagation bits \((g_i, p_i)\) of \( A + B + T \). From (3), when \( m = 2^n - 2^k - 1 \),

\[
T = 2^k g_k 2^n - 2^k - 1 - m = 2^k + 1
\]

Obviously, the binary representation of \( T \) is \(
\underbrace{00 \ldots 00}_{(n-k)\text{bit}} 100 \ldots 011^{(k)\text{bit}}
\).

In Fig. 2, the computation of \( A + B + T \) can be performed by \( A1 \) and \( A2 \) where \( A1 \) and \( A2 \) are used for lower-\( k \) bits and higher-\( n-k \) bits addition, respectively. Let \( T_{A1} = 00 \ldots 011^{(k)\text{bit}} \) and \( T_{A2} = 00 \ldots 011^{(k)\text{bit}} \), and the binary representations of \( A \) and \( B \) be

\[
\begin{aligned}
an_{1} \ldots a_{k} & b_{k} a_{k+1} \ldots a_{n} \quad \text{and} \quad bn_{1} \ldots b_{k} b_{k+1} \ldots b_{n}
\end{aligned}
\]

respectively. The operation of adder \( A1 \) and \( A2 \) can be regarded as

\[
\begin{aligned}
S_{A1} &= a_k \ldots a_{i} + b_k \ldots b_{i} + T_{A1} \\
S_{A2} &= a_{n-1} \ldots a_{k} + b_{n-1} \ldots b_{k} + T_{A2} + c_{\text{fin}}
\end{aligned}
\]

where \( c_{\text{fin}} \) is the carry-out bit of adder \( A1 \).

For \( T_{A1} \), one of the inputs of \( A1 \), every bit is “0” except the least significant bit. Thus, \( A1 \) can be treated as a \( k \)-bit adder with the lowest carry-in bit, which is exactly as same as the general binary adder. And the way pre-processing of \( T_{A1} \) is also similar with the general binary adder. The difference is that the lowest carry-in bit should be considered. Therefore, carry generation and carry propagation bits are

\[
\begin{aligned}
\left\{ \begin{array}{ll}
(g_0, p_0) &= (a_0 + b_0, a_0 \oplus b_0) \\
(g_i, p_i) &= (a_i b_i, a_i \oplus b_i) \\
\end{array} \right. \\
& i = 0, 1, 2, \ldots, k - 1
\end{aligned}
\]

For adder \( A2 \), it does not only add the constant \( T_{A2} \), but also the carry-out bit \( c_{\text{fin}} \) from adder \( A1 \). It can be regarded as a three-inputs adder with the lowest carry-in bit. The three inputs are \( a_{n-1} \ldots a_{k} \), \( b_{n-1} \ldots b_{k} \) and \( T_{A2} \) in binary. In this paper,
we reduce the number of inputs from three to two for adder $A_2$ by using Simple Carry Save Adder (SCSA). When $i = k, k + 1, \ldots, n - 1$, we can get $(g'_i, p'_k)$ for $a_i$ and $b_i$, firstly
\[
g'_i, p'_k = (a_i b_i, a_i \oplus b_i). \tag{11}
\]

And then $(g'_i, p'_k)$ is treated as the inputs of the second stage in SCSA. The second stage of SCSA generates the carry generation and carry propagation bits from $(g'_i, p'_k)$ and $T_{A_2}$. Actually, it is the carry saved addition of these two binary numbers, $p_{n-1} \cdots p_{k+1} p'_k$ and $g'_n \cdots g'_{k+1} g'_k$. Thus, the final outputs of pre-processing unit for adder $A_2$ are
\[
\begin{align*}
g_{i+1} &= g'_{i+1}, \\
p_{i+1} &= p'_{i+1} \tag{12}
\end{align*}
\]

From (10) and (12), all of the information required in the prefix computation is obtained. Furthermore, the carry-out bit of SCSA, $c_{SCSA}$, is required to compute the carry-out bit of $A + B + T$, $c_{out}$. It is calculated as
\[
c_{SCSA} = a_{n-1} b_{n-1} = g'_{n-1}. \tag{13}
\]

### B. Carry Generation Unit

In carry generation unit, the carries $c_i^T$ $(i = 1, 2, \ldots, n)$ of $A + B + T$ can be obtained with the carry generation and carry propagation bits from the pre-processing unit. Any existing prefix structure can be used to get the carries $c_i^T$ in this paper.

It is worth pointing out that the carry-out bit of SCSA in the pre-processing unit, as shown in (13), is not involved in the prefix computation. Instead, $c_{SCSA}$ combined with the carry-out bit of the prefix tree is required to determine the carry-out bit of $A + B + T$ (denoted as $c_{out}$). It means that $c_{SCSA}$ in this paper.

Theorem 1 means that $c_i^0$ can be determined from $c_i^1$ by simple logic operation. That is the foundation of the carry correction for the proposed modular adder. We present the procedure of the carry correction in our scheme based on Theorem 1 as following.

For the proposed modulo $2^n - 2^k - 1$ adder, $T = 2^k + 1$ and can be represented as $(0 \cdots 001) \cdots (0 \cdots 001)\{0 \cdots 001\}$ in binary. The computation of $A + B + T$ can be divided into two steps, $S_A = A + B + (0 \cdots 001)\cdots (0 \cdots 001)\{0 \cdots 001\}$ and $S_B = S_A + (0 \cdots 001)\cdots (0 \cdots 001)\{0 \cdots 001\}$. The two “1” bits in $T$’s binary representation can be regarded as the carry-in bits for adder $A_1$ and adder $A_2$ shown in Fig. 2, respectively. Correspondingly, the carry bits of $A + B$ can be obtained with twice carry corrections of $A + B + T$ based on Theorem 1. The first correction result is the carries of $A + B + (0 \cdots 001)\cdots (0 \cdots 001)\{0 \cdots 001\}$. The second correction result is the carries of $A + B$. Whether carry correction is performed or not depends on the carry-out bit of $A + B + T$, that is, $c_{out}$ in (14).

#### Carry Correction for Adder $A_1$

The carry correction unit is used to get the real carries $c_i^{{\text{real}}}$ for each bit needed in the final sum computation stage. In order to reduce the area, we get the carries of $A + B$ by correcting the carries of $A + B + T$ in the carry correction unit.

We first derive the relation of $c_i^0$ and $c_i^1$ $(i = 0, 1, 2, \ldots, n)$ in binary addition in Theorem 1, where $c_i^0$ and $c_i^1$ are the carry outputs of prefix tree when the lowest carry in is “0” and “1”, respectively.

**Theorem 1:** Let $c_i$ $(i = 0, 1, 2, \ldots, n)$ be the carry bits of an $n$-bit adder, and they will be propagated to the higher adjacent positions, $c_{n-1}$ be the lowest carry in (that is, $c_0 = c_{n-1}$), and $c_{out}$ be the final carry-out bit (that is, $c_{out} = c_{n-1}$). Assuming the carries for each bit be $c_i^0$ when $c_{in} = 0$ and the carries for each bit be $c_i^1$ when $c_{in} = 1$, we can get the relationship
\[
c_i^0 = f_{i+1} c_{i+1} \tag{14}
\]

where $0 < i < n - 1$.

Proof: Let $a_{n-1} a_{n-2} \cdots a_0 = a_i$ and $b_{n-1} b_{n-2} \cdots b_0$ be the binary representations of $A$ and $B$, respectively. Then, we have $p_i = a_i \oplus b_i$, $g_i = a_i b_i$, and $F_{A_1} = p_i \cdots p_0 b_0$.

According to the parallel prefix algorithm, we have
\[
c_{i+1} = G_{i+1} + T_{i+1} = c_{in} + 1
\]

which can be rewritten as
\[
\begin{align*}
c_{i+1}^0 &= G_{i+1}^0 + c_{in}, \\
c_{i+1}^1 &= G_{i+1}^1 + c_{in} \tag{15}
\end{align*}
\]

If $P_{i+1} = 1$, then $a_i \neq b_i$, which yields $g_i = a_i b_i = 0$ and $G_{i+1} = 0$. Thus, we have $c_{i+1} = c_{in}$. That is, $c_{i+1}^0 = 0$, $c_{i+1}^1 = 1$.

If $P_{i+1} = 0$, it means that $c_{in}$ can’t be propagated to $c_{i+1}$. Hence, $c_{i+1} = G_{i+1}$, which is irrelevant with $c_{in}$. That means $c_{i+1}^0 = c_{i+1}^1 = c_{i+1}$.

From (10) and (14), all of the information required in the prefix computation is obtained. Furthermore, the carry-out bit of SCSA, $c_{SCSA}$, is required to compute the carry-out bit of $A + B + T$, $c_{out}$. It is calculated as
\[
c_{SCSA} = a_{n-1} b_{n-1} = g'_{n-1}. \tag{13}
\]

Theorem 1 means that $c_i^0$ can be determined from $c_i^1$ by simple logic operation. That is the foundation of the carry correction for the proposed modular adder. We present the procedure of the carry correction in our scheme based on Theorem 1 as following.

For the proposed modulo $2^n - 2^k - 1$ adder, $T = 2^k + 1$ and can be represented as $(0 \cdots 001) \cdots (0 \cdots 001)\{0 \cdots 001\}$ in binary. The computation of $A + B + T$ can be divided into two steps, $S_A = A + B + (0 \cdots 001)\cdots (0 \cdots 001)\{0 \cdots 001\}$ and $S_B = S_A + (0 \cdots 001)\cdots (0 \cdots 001)\{0 \cdots 001\}$. The two “1” bits in $T$’s binary representation can be regarded as the carry-in bits for adder $A_1$ and adder $A_2$ shown in Fig. 2, respectively. Correspondingly, the carry bits of $A + B$ can be obtained with twice carry corrections of $A + B + T$ based on Theorem 1. The first correction result is the carries of $A + B + (0 \cdots 001)\cdots (0 \cdots 001)\{0 \cdots 001\}$. The second correction result is the carries of $A + B$. Whether carry correction is performed or not depends on the carry-out bit of $A + B + T$, that is, $c_{out}$ in (14).

#### Carry Correction for Adder $A_1$

Since the binary representation of $T$ is $(0 \cdots 001) \cdots (0 \cdots 001)$, $c_i^0$ can be regarded as the carries of $A + B + T - 1 + c_{in}$ and $c_{in} = 1$.

Therefore, $c_i^1$ can be modified with Theorem 1 to determine the carry bits $c_{i+1} - (i = 0, 1, 2, \ldots, n - 2)$ of $A + B + T - 1$, that is
\[
c_i^0 = P_{i+1} c_{i+1} \tag{16}
\]

One point must be paid attention to perform (15). The lowest propagation bit in $P_{i+1} = 0$, is not equal to that in (10). Actually, it is equal to $a_0 \oplus b_0$.

According to Theorem 1, the carries of $A + B + T$ is corrected under the condition of $c_{in} = 0$. We can use a 2-to-1 Multiplexer (MUX) to perform the operation. For this MUX, $c_{out}$ is the control signal, while $c_i^T$ and $c_i^{T-1}$
are input signals. And the output is the result of the first correction, denoted as 
\[ c_{i+1} = c_{i+1}^{\text{out}} + c_{i+1}^{T} + c_{i+1}^{\text{out}} P_{i} \] 
(16)

### Carry Correction for \( A2 \)

From (16), \( c_{i+1}^{\text{out}} \) (\( i = 0, 1, 2, \ldots, n - 2 \)) is the carry information of \( A + B + T \). Let the carry bits of the second correction be \( c_{i}^{\text{real}} \). Similar to the first correction, \( c_{i}^{\text{real}} \) is the carry of \( A + B + T - 1 \) (that is, \( A + B \)) when \( c_{\text{out}} = 0 \). Otherwise, \( c_{i}^{\text{real}} \) is the carry of \( A + B + T \). That is, \( c_{i}^{\text{real}} \) is the final carry information needed in sum computation unit.

When \( i = 1, 2, \ldots, k, T = 2^{k} + 1 \). The bit “1” in \( T_{A2} \) will not affect \( c_{i}^{T} \). Hence,
\[ c_{i}^{\text{real}} = c_{i}^{T} \] 
(17)

When \( i = k + 1, \ldots, n - 1 \), the inputs of adder \( A2 \) in Fig. 2 are \( P_{n-1}^{k+1} \), \( P_{n}^{k} \), \( P_{k} \), and \( g_{n-2} \cdots g_{k+1} g_{k} \). And the carry-in bit is the carry-out bit of adder \( A1 \), that is, \( c_{k}^{T} \). Considering the least significant bit of \( g_{n-2} \cdots g_{k+1} g_{k} \) is “1”, we can treat the operation of adder \( A2 \) as the addition of two inputs, \( P_{n-1}^{k+1} P_{k}^{k} \), \( g_{n-2} \cdots g_{k+1} g_{k} \), with the lowest carry-in bit “1”.

Thus, the results and carry information of (a), (b) in (18) are identical
\[ \begin{align*}
\{ & P_{n-1}^{k+1} + P_{k}^{k} + g_{n-2} \cdots g_{k+1} g_{k} + (0) \cdots c_{i}^{T} \\
& \quad (n-k) \text{bit} \\
& P_{n-1}^{k+1} + P_{k}^{k} + g_{n-2} \cdots g_{k+1} g_{k} c_{i}^{T} + (0) \cdots (n-k) \text{bit} \\
\end{align*} \] 
(18)

Substituting (19) into (21), we get
\[ c_{i}^{\text{real}} = c_{i+1}^{T} (c_{\text{out}} + P_{i} 0) \left( P_{i} + P_{i+1} + P_{i} c_{k}^{T} + P_{i} c_{k}^{T} \right) \] 
(20)

When \( i = k + 1, k + 2, \ldots, n - 2 \), according to Theorem 1 and (16), the carries after the second carry correction are
\[ c_{i}^{\text{real}} = \left( P_{i} 0 + c_{\text{out}} \right) c_{i+1}^{T} = \left( P_{i} + P_{i+1} + P_{i} c_{k}^{T} + P_{i} c_{k}^{T} \right) c_{i+1}^{T} + c_{i+1}^{T} (c_{\text{out}} + P_{i} 0) \left( P_{i} + P_{i+1} + P_{i} c_{k}^{T} + P_{i} c_{k}^{T} \right) \] 
(21)

Thus, we can get the carries of \( A + B \) by modifying the carries \( c_{i}^{T} \) (\( i = k + 1, \ldots, n - 1 \)) of adder \( A2 \) with Theorem 1. Combined with the final carry-out bit of \( A + B + T \), \( c_{\text{out}} \), the carries \( c_{i}^{\text{real}} \) required by the proposed modular adder are determined.

Since the second carry correction is performed under the condition that the lowest carry-in bit of adder \( A2 \) is a constant “1”, the propagation bits used in the carry correction unit should be computed by \( P_{n-1}^{k+1} P_{k}^{k} \) and \( g_{n-2} \cdots g_{k+1} g_{k} c_{i}^{T} \). From the above analysis, it is shown that the difference between these two additions in (18) is that the least significant bits, “1” for \( g_{n-2} \cdots g_{k+1} g_{k} \) in (18) (a) and \( c_{i}^{k} \) for \( g_{n-2} \cdots g_{k+1} g_{k} c_{i}^{T} \) in (18) (b). The propagation carry information can be computed from (11) and (12). Let \( p_{k}^{i} \) be the group propagate carries of (18) (b), we have
\[ \begin{align*}
\{ & p_{k}^{i} = p_{k}^{i} + c_{k}^{T} = P_{k} + c_{k}^{T} \\
& \quad i = k \\
& p_{k}^{i} - p_{k} \quad i = k + 1, \ldots, n - 1. \\
\end{align*} \] 
(19)

Let \( P_{k}^{i} \) be the group propagate carries, then
\[ P_{k}^{i} = P_{k}^{i} 1 \] 
(20)

According to (16), (17), (23) and (24), the carry bits required by the proposed modular adder are determined as shown in (25), at the bottom of the page.
Let
\[
\begin{align*}
  z_1 &= \frac{P_{k-1} \odot (p_k + c^T_k)}{z_2 = \frac{P_{k+1} + P_{k-1} \odot (p_k + c^T_k)}}.
\end{align*}
\]
(26)
Then
\[
\begin{align*}
  c^\text{real}_{i+1} &= \begin{cases} 
  \left(c^T_{i+1} \oplus (c^\text{out} + \overline{P_{0}}) \right) & i = 0, 1, \ldots, k - 1 \\
  \left(c^T_{i+1} \oplus \overline{c^\text{out}} \right) & i = k, k + 1, \ldots, n - 2.
\end{cases}
\end{align*}
\]
(27)
From the unit-gate evaluation model, the delay of computing \(c^\text{real}_{i+1}\) \(i, 0, 1, \ldots, k - 1\) is \(2\tau_g \) when \(i = 1, 2, \ldots, k\), which is identical to the delay of a prefix computation unit. It is shown from Fig. 2 that the pre-processing units of the proposed modular adder guarantees that \(c^\text{real}_{i+1}\) \(i = k, k + 1, \ldots, n - 1\) at least \(2\tau_g\) for most prefix structures.

If \(c^T_k\) is determined before \(c^T_i\) \(i = k + 1, \ldots, n - 1\) no less than two stages prefix computation, the delay of computing \(z_1\) and \(z_2\) in (26) is the delay of sum of one XOR, one AND, and one OR gate. That means the output time of \(c^T_i\) \(i = k + 1, \ldots, n - 1\) is identical with that of \(z_1\) and \(z_2\) in (26). Thus, there is also no extra delay.

If \(c^T_k\) is determined before \(c^T_i\) \(i = k + 1, \ldots, n - 1\) only less than one stage prefix computation delay, the delay of computing \(z_1\) and \(z_2\) should be reduced to at most one prefix computation delay through special pre-processing to eliminate the possible extra delay. In order to achieve this purpose, \(c^T_k\) can be used as the selection signal for the MUX. Meanwhile, \(z_1\) and \(z_2\) can be pre-computed and used as the inputs of the MUX. Let \(z^0_k\) and \(z^1_k\) be the value of \(z_1\) and \(z_2\) when \(c^T_k = 0\) respectively. Similarly, let \(z^0_i\) and \(z^1_i\) be the value of \(z_1\) and \(z_2\) when \(c^T_k = 1\) respectively. We get
\[
\begin{align*}
  \left\{ 
  \begin{array}{l}
    z^0_1 = P_{k-1,0} + \overline{p_k} \\
    z^1_1 = P_{k-1,0} + p_k \\
    z^0_2 = P_{k+1}(P_{k-1,0} + \overline{p_k}) \\
    z^1_2 = P_{k+1}(P_{k-1,0} + p_k)
  \end{array}
\right.
\end{align*}
\]
(28)
and
\[
\begin{align*}
  \left\{ 
  \begin{array}{l}
    z^0_1 - \frac{c^T_k}{2} z^0_2 + c^T_k z^1_2 = c^T_k(P_{k-1,0} + \overline{p_k}) + c^T_k(P_{k-1,0} + p_k) \\
    z^1_2 = c^T_k(\overline{c^\text{out}} + \overline{P_{k+1}} + P_{k-1,0} + p_k) + c^T_k(P_{k+1}(P_{k-1,0} + \overline{p_k})) + c^T_k(P_{k+1}(P_{k-1,0} + p_k))
  \end{array}
\right.
\end{align*}
\]
(29)
Thus, we can get the carry information that will be used in the sum computation unit of the proposed modular adder.

D. The Sum Computation

Generally, the sum computation is as same as that in prefix-based binary adder. However, \(c^\text{real}_{i+1}\) is the correction result when \(c^\text{out}\) is taken into account. That is, if \(c^\text{out} = 0\), \(c^\text{real}_{i+1}\) is the carry bit of \(A + B\). Otherwise, it is the carry bit of \(A + B + T\). Thus, the partial sum bits of \(A + H\) and \(A + H + T\) are both required in the final sum computation. Let \(p^0_k\) and \(p^1_i\) \((i = 0, 1, \ldots, n - 1)\) be the partial sum bits of \(A + B\) and \(A + B + T\) respectively. Note that \(p^1_i\) \((i = 0, 1, \ldots, n - 1)\) has been determined in the pre-processing unit (that is, \(p^1_i = p_i\)). Besides, \(p^0_k \neq p^1_i\) just when \(i = 0\) and \(k\). Consequently
\[
\begin{align*}
  &\begin{cases} 
    p^0_k = P_{0,0} + p_k, i = 0 \\
    p^0_k = P_{k+1}, p^1_i = p_k \quad i = k \\
    p^1_i = p^1_i \quad i = 1, \ldots, k - 1, k + 1, \ldots, n - 1.
  \end{cases}
\end{align*}
\]
(30)
Hence
\[
\begin{align*}
  s_0 &= c^\text{real}_{0} \oplus \overline{c^\text{out}}p^0_0 + c^\text{out}\overline{p^0_0} - c^\text{out}p^0_0 - c^\text{out}p^0_0 + c^\text{out} \oplus \overline{p^0_0} \quad i = 0 \\
  s^i_k &= c^\text{real}_{i} \oplus \overline{c^\text{out}}p^0_k + c^\text{out}p^0_k \quad i = k \\
  s^i_k &= c^\text{real}_{i} \oplus p^1_i \quad i = 1, \ldots, k - 1, k + 1, \ldots, n - 1.
\end{align*}
\]
(31)
(32)
When \(i = 1, \ldots, k - 1, k + 1, \ldots, n - 1\)
\[
s^i_k = c^\text{real}_{i} \oplus p^1_i.
\]
(33)
At last, the sum bits are
\[
\begin{align*}
  s^0_0 &= c^\text{out} + p^0_0 + c^\text{out} + \overline{p^0_0} \\
  s^i_k &= c^\text{real}_{i} \oplus c^\text{out} \oplus p^0_k \quad i = k \\
  s^i_k &= c^\text{real}_{i} \oplus p^1_i \quad i = 1, \ldots, k - 1, k + 1, \ldots, n - 1.
\end{align*}
\]
(34)
In (34), \(c^\text{out} \oplus p^0_k\) and \(c^\text{real}_{i}\) can be obtained at the same time. Therefore, there is no extra delay compared with other sum computation units.

E. Design Example

The VLSI implementation structure of modulo \(2^{8} - 2^{4} - 1\) adder based on the proposed scheme is shown in Fig. 3(a). Fig. 3(b) illustrates the function of each module.

— Pre-processing Unit

The pattern “\(0\)” in Fig. 3 is the pre-processing unit and used to generate carry generation and carry propagation bits for the following prefix computation. Since there are fixed “1” inputs at the 1st and the 4th places, the patterns “\(\overline{0}\)” and “\(\overline{1}\)” are used for this special situations. The pattern “\(\overline{0}\)” does not cost any resource in unit-gate model. The computations of these patterns correspond to (10), (11) (12).

— Prefix Computation

The pattern “\(\ast\)” is the prefix computation unit. In this example, the Sklansky prefix tree is used and there are 11 prefix computation units, which corresponds to (4). The delay of “\(\ast\)” is determined by its’ carry generation path which is one OR gate and one AND gate. However, the pattern “\(\ast\)” in the final stage of prefix tree is not needed to compute propagation bits.

— The Computation of \(c^\text{out}\)

The \(c^\text{out}\) is computed by pattern “\(\odot\)” in Fig. 3. According to (14), \(c^\text{out} = c^\text{OUT} + G_{n-1,\ast} + P_{n-1,\ast}c^T_{n-1} + c^\text{OUT} + G_{n-1,\ast} + P_{n-1,\ast}c^T_{n-1}\) can be computed concurrently. Then, we can get \(c^\text{out}\) after an OR gate. Thus, the delay of \(c^\text{out}\) computation will not exceeding the delay of pattern “\(\ast\)” and there is no extra delay. In order to minimize the delay of “\(\ast\)” in the final stage of prefix tree is not needed to compute propagation bits.
least one OR gate delay. In (14), $c_{SCSA}$ is computed in pre-processing firstly. Meanwhile, the delay of $P_{n-1;i}$ is always smaller than that of $G_{n-1;i}$ in prefix tree. Thus, we can compute $c_{SCSA} + G_{n-1;i}$ firstly if $G_{n-1;i}$ is obtained before $c_{i}^{T}$. Otherwise, we can compute $c_{SCSA} + P_{n-1;i}c_{i}^{T}$ firstly. When the last one, $c_{i}^{T}$ or $G_{n-1;i}$, arrived, only one OR gate is needed to compute the final value of $c_{out}$. That is, the delay is $\tau_g$ if the value of $l$ is selected properly. In this example, $i = k$.

— Carry Correction Unit

The pattern “$\bigcirc$” in Fig. 3 performs the computation correspond to (27). In this example, 7 correction operators are used. From (27), there are three different situations, that is $i = 0$, 1, ..., $k - 1$, $i = k$ and $i = k + 1$, ..., $n - 2$. The $P_{k;0}$, $z_1$ and $z_2$ can be computed by independent modules. The pattern “$\square$” and “$\boxsupset$” in Fig. 3 is used to compute $P_{k;0}$, $z_1$ and $z_2$ in (27). In this example, $c_{i}^{T}$ is computed out before $c_{i}^{T}(i = k + 1, \ldots, n - 1)$ with two prefix computation stages. Hence, we can get $z_1$ and $z_2$ without extra delay by using (26). In the worst case, the group propagation bits required in (26) are needed to be computed one by one from $p_i(i = 0, 1, \ldots, n - 2)$. However, the extra components for computing these group propagation bits can be removed when the group propagation bits exist in prefix structure.

— Sum Computation Unit

The pattern “$\bigoplus$” in Fig. 3 is used for performing the sum computation according to (34). As a matter of fact, this operator is the logic XOR operation. The pattern “$\bigoplus'$” in Fig. 3 is a modified XOR operator, one of its inputs is inverted. Because the computation of $c_{out} \oplus p_{i}^{T}$ in (34) can be performed with carry correction simultaneously, only one XOR operations are required to perform the sum computation and no extra delay is introduced.

— Numerical Example

For example, for modulo 239 (that is, $n = 8$ and $k = 4$) addition, $T = 2^4 + 1 = 17$. If $A = 215$ and $B = 177$, the result of the modular addition is 153. According to (10), (11) and (12), pre-processing results are

\[
\{g_0, g_1, g_2, g_3\} = \{1, 0, 0, 0\}, \{p_0, p_1, p_2, p_3\} = \{1, 1, 1, 0\}.
\]

\[
\{g_4, g_5, g_6, g_7\} = \{0, 1, 0, 0\}, \{p_4, p_5, p_6, p_7\} = \{0, 1, 1, 0\}.
\]

Then, by using prefix tree and (13), we have

\[
\{c_1^T, c_2^T, c_3^T, c_4^T, c_5^T, c_6^T, c_7^T\} = \{1, 1, 1, 0, 0, 1, 1\}
\]

and

\[
c_{out} = c_7^T + g_7^T = 1.
\]

From (25), we can get carry correction results

\[
\{c_1^{real}, c_2^{real}, c_3^{real}, c_4^{real}, c_5^{real}, c_6^{real}, c_7^{real}\} = \{1, 1, 1, 0, 1, 1, 1\}.
\]

Finally, the modulo addition results can be computed by (34),

\[
{g_0, g_1, g_2, g_3, g_4, g_5, g_6, g_7} = \{1, 0, 0, 1, 1, 0, 0, 1\}.
\]

That is the binary representation of 153.
This example shows the detailed design of modulo $2^n - 2^k - 1$ adder based on the proposed algorithm with the Sklansky prefix tree. There are two special measures in the proposed scheme are used to eliminate the possible extra delay. The first one is the computation of $c_{out}$ in (14) which shows the way of eliminating the delay. In fact, it is easy to satisfy requirements of (14) for an adder based on prefix structure. The second one is the pre-processing of temporary variables in carry correction. In the worst case, $p_i (i = 0, 1, \ldots, n - 2)$ are needed to determine the group propagate bits required in (25) by using independent modules. Nevertheless, the special logical resource for computing group carry information can always be reduced according to the prefix structure used in the proposed modular adder.

IV. PERFORMANCE ANALYSIS AND COMPARISON

A. Performance Analysis and Comparison Based on Unit-Gate Model

According to (5), the delay of prefix tree is always determined by the path of carry generation units which is $2 \tau_g$. However, the delay of the pre-processing units and carry generation units at the first level of prefix tree can be reduced to $3 \tau_g$. Let $x_i, y_i (i = 0, 1, \ldots, n - 1)$ be the inputs of pre-processing units and $g_i, p_i (i = 0, 1, \ldots, n - 1)$ be the outputs of pre-processing units. If $p_i$ is computed by

$$p_i = a_i \oplus b_i = \overline{a_i b_i} (a_i + b_i)$$

we can get

$$G^1_{i+1:} = g_i + p_{i+1} g_i = a_i b_{i+1} + a_i b_{i+1} g_i = a_i b_{i+1} + b_i g_i + g_i = g_{i+1} + (a_i + b_{i+1}) g_i.$$

Obviously, the critical path delay of pre-processing and the first level prefix computation is $3 \tau_g$. Meanwhile, we can get $a_i b_i$ and $a_i + b_i$ in the computation procedure of (36). As result, there is no extra area.

The delay of carry correction units and sum computation units are both $2 \tau_g$. As for prefix operation, its delay depends on the adopted prefix structure. According to the above analysis, the critical path delay of the proposed modulo $2^n - 2^k - 1$ adder is the sum of the delay of prefix structure and 7 unit gates. That is

$$\tau = \tau_p + 7 \tau_g.$$
TABLE III
THE AREA AND DELAY COMPARISON BASED ON UNIT-GATE MODEL

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Area ((\mu m^2))</th>
<th>Delay ((t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11]</td>
<td>((4n-7)\log_2(n+1)+1n+4)</td>
<td>(2\log_2 n+7)</td>
</tr>
<tr>
<td>[7]</td>
<td>(1.5(n-1)\log_2 (n+1)+1.5n\log_2 n+16n-4)</td>
<td>(2\log_2 n+8)</td>
</tr>
<tr>
<td>[10]</td>
<td>(1.5\log_2 n+14n+6)</td>
<td>(4\log_2 n+8)</td>
</tr>
<tr>
<td>[20]</td>
<td>(1.5n\log_2 n+1.5n\log_2 (n-1)+2n(n-1)+11n-2)</td>
<td>(2\log_2 n+5)</td>
</tr>
<tr>
<td>[8]</td>
<td>((5n-1)\log_2 n+11.5n+1)</td>
<td>(2\log_2 n+7)</td>
</tr>
<tr>
<td>Proposed</td>
<td>(1.5n\log_2 n+13n-5k-8)</td>
<td>(2\log_2 n+7)</td>
</tr>
</tbody>
</table>

ASIC (Application Specific Integrated Circuit) synthesis they are also implemented based on Sklansky prefix tree. Meanwhile, the analysis of [10] is under the assumption that there are only two “1” in \(T\)’s binary representation.

In [8], a new number representation method is adopted to simplify modulo addition. Conversion from binary to its special representation bears no cost. However, its addition results are in this special number representation format. Extra area and delay should be used to perform the conversion from this format to binary number representation or all operations, such as addition and multiplication, should be performed in this number representation format in RNS-based system. In order to perform comparison without the conversion effect, the conversion from its special number format to binary representation or all operations, such as addition and multiplication, should be performed in this number representation format in RNS-based system. In order to perform comparison without the conversion effect, the conversion from its special number format to binary representation is not included in the analysis and comparisons. Table III shows that its area is similar with that of [20] and the delay is similar with that of [11].

The modulo \(2^n - 2^n - 2 - 1\) adder proposed by [20] is the special case of our scheme. Since the position of “1” in \(T\) of modulo \(2^n - 2^n - 2 - 1\) adder is fixed, some optimizations can be done so as to reduce the delay of pre-processing module. Thus, the total delay of this modular adder is \((2\log_2 n + 5)\tau_c\).

Table III shows that the largest area is needed in [7] and the smallest is needed in our scheme. Meanwhile, Table III also shows that the fastest speed is [20] and the slowest is [10]. However, the unit-gate model is just a reference in performance analysis. In practice, different architecture may have different ability in tradeoff between area and delay.

In Section IV-B, we will implement all scheme mentioned in Table III and perform detailed comparison based on the common used synthesis tool, DC.

B. Performance Analysis and Comparison Based on Design Compiler

In order to get more accurate performance evaluation, we design the proposed modulo \(2^n - 2k - 1\) adder with Sklansky prefix tree and the other modulo adders mentioned in Table III with VHDL. Then, we use DC to get area and delay performance. The version of DC is E-2010.12-SP5-2 for LINUX. And we use its TOPOGRAPHICAL mode to get more accurate wire load model. Then, these designs are synthesized with the Taiwan Semiconductor Manufacturing Company (TSMC) 0.13 \(\mu m\) logical library. Meanwhile, the TSMC 0.13 \(\mu m\) physical library is used to get more accurate area and timing evaluation in logical synthesis procedure. For comprehensive comparison, we first design these modular adders in Table III for \(n = 4, 6, 12\) at two cases, \(k = 1\) and \(k = n - 2\). Then, we design our scheme for \(k = 2, 3, 4, 5\) when \(n = 8\) to get the performance change with the different value of \(k\). Two different optimization approaches are used in the following ASIC synthesis procedures.

The first optimization approach is that each design is recursively optimized until they achieved a fastest operating frequency without timing violation and the value of slack is zero. The timing constraint step is 0.01 ns in recursive optimization procedure.

Table IV is the synthesis results of area and delay for our scheme when \(n = 8\) and \(k\) varies from 1 to 6. The results in Table IV show that the delay and area decrease with the increase of \(k\) in value. They also indicate that the area and delay is not changing in a linear fashion with the variation of \(k\). However, the ASIC synthesis results in Table IV reveal the changing trend in delay and area with the variation of \(k\).

Table V, Table VI, and Table VII are the synthesis results of area and delay for these modular adders when \(n = 8, 12\), and 12, respectively. The values in the rightmost column of Table V, Table VI and Table VII are the “area*delay” ratio to ELMMA. In our design, the propagation bits needed in carry correction unit are calculated by independent modules.

Table V, Table VI and Table VII show that [7] has the largest area and [10] has the largest delay at most cases. As for the modular adder proposed by [20], some optimization for the delay can be done because it just works at a special case, \(k = n - 2\). Thus, the delay of the proposed modular adder is a little worse than [20] in theory. Furthermore, the overall performance, “area*delay”, of the proposed modular adder have similar performance with [20] when \(n = 6\) and 8. Although the “area*delay” performance of the proposed modular adder is
TABLE VI
ASIC SYNTHESIZED RESULTS FOR TIMING OPTIMIZATION II \(n = 8\)

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Area ((\mu m^2))</th>
<th>Delay (ns)</th>
<th>(A^<em>D^</em>)</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11]</td>
<td>345.50</td>
<td>0.82</td>
<td>283.31</td>
<td>100.00</td>
</tr>
<tr>
<td>[11]</td>
<td>267.50</td>
<td>0.80</td>
<td>214.00</td>
<td>100.00</td>
</tr>
<tr>
<td>[7]</td>
<td>450.00</td>
<td>0.83</td>
<td>373.50</td>
<td>131.83</td>
</tr>
<tr>
<td>[7]</td>
<td>280.00</td>
<td>0.76</td>
<td>212.80</td>
<td>99.44</td>
</tr>
<tr>
<td>[10]</td>
<td>355.00</td>
<td>0.91</td>
<td>323.05</td>
<td>114.03</td>
</tr>
<tr>
<td>[10]</td>
<td>287.75</td>
<td>0.88</td>
<td>253.22</td>
<td>118.33</td>
</tr>
<tr>
<td>[20]</td>
<td>249.50</td>
<td>0.72</td>
<td>179.64</td>
<td>83.94</td>
</tr>
<tr>
<td>[8]</td>
<td>371.75</td>
<td>0.83</td>
<td>308.55</td>
<td>108.91</td>
</tr>
<tr>
<td>[8]</td>
<td>251.21</td>
<td>0.81</td>
<td>203.51</td>
<td>95.10</td>
</tr>
<tr>
<td>Proposed (k=1)</td>
<td>353.50</td>
<td>0.82</td>
<td>289.87</td>
<td>102.32</td>
</tr>
<tr>
<td>Proposed (k=6)</td>
<td>254.00</td>
<td>0.70</td>
<td>177.80</td>
<td>83.08</td>
</tr>
</tbody>
</table>

\(A^*D^* = \text{Area} \times \text{Delay}\)

TABLE VII
ASIC SYNTHESIZED RESULTS FOR TIMING OPTIMIZATION III \(n = 12\)

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Area ((\mu m^2))</th>
<th>Delay (ns)</th>
<th>(A^<em>D^</em>)</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11]</td>
<td>477.25</td>
<td>0.96</td>
<td>458.16</td>
<td>100.00</td>
</tr>
<tr>
<td>[11]</td>
<td>414.00</td>
<td>0.95</td>
<td>393.30</td>
<td>100.00</td>
</tr>
<tr>
<td>[7]</td>
<td>710.00</td>
<td>1.00</td>
<td>710.00</td>
<td>154.97</td>
</tr>
<tr>
<td>[7]</td>
<td>635.00</td>
<td>1.03</td>
<td>654.05</td>
<td>166.30</td>
</tr>
<tr>
<td>[10]</td>
<td>570.00</td>
<td>1.15</td>
<td>565.50</td>
<td>143.07</td>
</tr>
<tr>
<td>[10]</td>
<td>365.00</td>
<td>1.06</td>
<td>386.90</td>
<td>98.37</td>
</tr>
<tr>
<td>[20]</td>
<td>308.25</td>
<td>0.83</td>
<td>255.85</td>
<td>65.05</td>
</tr>
<tr>
<td>[8]</td>
<td>540.50</td>
<td>0.96</td>
<td>518.88</td>
<td>113.25</td>
</tr>
<tr>
<td>[8]</td>
<td>481.50</td>
<td>0.94</td>
<td>452.52</td>
<td>115.06</td>
</tr>
<tr>
<td>Proposed (k=1)</td>
<td>522.25</td>
<td>0.97</td>
<td>506.58</td>
<td>110.57</td>
</tr>
<tr>
<td>Proposed (k=10)</td>
<td>433.75</td>
<td>0.83</td>
<td>360.01</td>
<td>91.54</td>
</tr>
</tbody>
</table>

\(A^*D^* = \text{Area} \times \text{Delay}\)

The second optimization approach is that these designs with the same value of \(n\) are optimized for area under a timing constraint. Meanwhile, in order to get better area optimization, these target delays for different \(n\) are set to the double of the max value in the third column in Table VI, VII, respectively. That is, the target delay for all designs is set to 1.72 ns when \(n = 8\), 1.82 ns when \(n = 8\), and 2.3 ns when \(n = 12\). Meanwhile, the set_max_area parameter in DC is set to zero for all designs. The difference from timing optimization approach is that we first optimize area and followed by delay. Table VIII is the synthesis results for area optimization. It shows that the maximum area is needed in [7] and the maximum delay is needed in [10] at most cases. Our scheme has similar performance in area and delay with [20] when \(n = 6, 8\) with \(k = n - 2\). When \(n = 12\) with \(k = 10\), [20] has the best performance in area because of its special design for only one case. Table VIII also shows that our design has little worse in area to [11] when \(k = 1\). This is because the proposed modular adder needs more carry correction processing units when \(k = 1\) and these pre-processing units are implemented independently. However, the word lengths in common RNS-based applications are usually shorter than 8 bits. Meanwhile, the proposed adder has better performance in delay, especially when \(k = n - 2\).

V. CONCLUSION

In this paper, a new class of modulo \(2^n - 2^k - 1\) adder is proposed. The proposed structure is consisted of four units, the pre-processing, the carry computation, the carry correction and the sum computation unit. The performance analysis and comparison show that the proposed algorithm can construct a new class of general modular adder with better performance in delay or “area*delay”. It has some main features as following:

The way using twice carry corrections improves the performance of area and timing in VLSI implementation and reduces the redundant units for parallel computation of \(A + B + T\) and \(A + B\) in the traditional modular adders.

Any existing prefix tree can be used in this structure. That means fine tradeoff property between area and delay for the proposed scheme. The synthesis results also show that our scheme can be optimized to work at faster operation frequency.

Furthermore, the modulus with the form of \(2^n - 2^k - 1\) facilitate the construction of a new class of RNS with larger dynamic and more balanced complexity among each residue channel. The work of this paper provides an alternative scheme of modular adder design for this type of RNS.

REFERENCES


