A One-Step Modulo $2^n+1$ Adder Based on Double-lsb Representation of Residues

Ghassem Jaberipur

Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran

Abstract

Efficient modulo $2^n+1$ adders are desirable for computer arithmetic units based on residue number systems (RNS) with the popular moduli set \{2$^{n-1}$, 2$^n$, 2$^{n+1}$\}. Regular $n$-bit ripple-carry adders or their fast equivalents are suitable for modulo $2^n$ addition. But for the other two moduli a correcting increment/decrement step besides the primary $n$-bit addition is normally required. Several design efforts have tried to reduce the latency of the correcting step to a small delay not depending on the word length $n$, leading to one-step modular addition schemes. These include the use of alternative encoding of residues (e.g., diminished-1 representation of modulo $2^n+1$ numbers), customized (vs. generic) adders (e.g., specialized parallel prefix adders), or compound adders. In this paper we investigate alternative modulo $2^n+1$ addition schemes, focus on generic one-step adder designs, and use the double-lsb representation of modulo $2^n+1$ numbers. In a generic modular adder, the central abstract $n$-bit adder may be replaced by any concrete adder architecture meeting the designer’s prescribed measures in time, area and power consumption.

Keywords: Residue number system, Modulo $2^n+1$ addition, Double-lsb representation, Diminished-1 number representation, Parallel prefix adders, Generic adders.

1. Introduction

Residue number systems (RNS) and the related arithmetic units are popular in many digital signal processing applications where most computations are restricted to multiplication, addition and subtraction [1]. Application areas, besides general RNS arithmetic, as noted in [2] include:

- Fast number theoretic transforms
- Discrete Fourier transform
- Digital filters

A residue number system $\Psi$ is characterized by $k$ integer moduli $m_{k-1}, \ldots, m_1, m_0$. An integer value $x$ ($\alpha \leq x \leq \beta$) is uniquely represented by $(x_{k-1}, x_1, x_0)$, where $x_i = x \mod m_i$ ($0 \leq i \leq k - 1$), and $[\alpha, \beta]$ is the dynamic range of $\Psi$, whose cardinality is $M_{\Psi} = \beta - \alpha + 1$. To maximize $M_{\Psi}$, the moduli are chosen to be pair-wise prime, such that $M_{\Psi} = m_{k-1} \times \ldots \times m_1 \times m_0$.

To compute $x \bullet y$, where $\bullet$ is an arithmetic operator (e.g., $+$ or $\times$), one simply performs modular $\bullet$ on each of the $k$ residues of $x$ and $y$ in parallel as in Equation (1), where subscript $m_i$ ($0 \leq i \leq k - 1$) stands for modulo $m_i$ operation, $x = (x_{k-1}, \ldots, x_1, x_0)$, and $y = (y_{k-1}, \ldots, y_1, y_0)$.

$$x \bullet y = (x_{k-1} \bullet y_{k-1})_{m_{k-1}} \ldots (x_1 \bullet y_1)_{m_1} (x_0 \bullet y_0)_{m_0} \quad (1)$$

RNS addition, as also the basis for subtraction and multiplication, is composed of adding the two residues per each of the moduli followed by a mod operation. Therefore the latency of RNS addition would at best be in the order of $\lceil \log (\max (m_{k-1}, \ldots, m_1, m_0)) \rceil$, which is considerably less than that of same operation on full-words $x$ and $y$, provided that the difference between minimum and maximum moduli is minimal. For higher speed, however, the moduli with simpler mod operation are preferred.
The first choice is 2^n leading to fastest mod operation, which is actually performed by ignoring the carry-out signal of the residue adder and keeping the sum. Next popular choices have primarily been 2^n±1 [3, 4, 5, and 6]. The three moduli are pair-wise prime and the required dynamic range can be met by choosing the proper value for n. Unlike the 2^n case, the mod operation for moduli 2^n±1 may involve substantive overhead in terms of hardware and latency, but yet much less than similar operation for other moduli. Previous research on modulo 2^n±1 adders has shown that modulo 2^n±1 adder is more complex than that of the conjugate modulo [7]. Therefore, any performance gain in modulo 2^n±1 adder will enhance the efficiency of the popular {2^n, 2^n±1} RNS.

In this paper we focus on modulo 2^n+1 addition. Besides the general RNS applications, listed above, Efstathiou et. al. [8] categorize alternative applications of modulo 2^n+1 addition as:

- Cryptography
- Pseudorandom number generation
- Convolution computation

Zimmermann [3] offers a thorough analysis of different possible modulo 2^n±1 arithmetic implementations that had been appeared in the literature, where modulo 2^n+1 addition is either performed in two addition cycles or with two parallel adders. Two parallel-prefix modulo 2^n+1 addition schemes have been proposed in [6 and 7]. The adder architecture in both contributions is based on end-around carry increment after the main addition cycle. However the end-around carry increment operation is fused in the final carry computation logic. The latter design, although successful in using a single n-bit adder, is not a generic design in a sense to allow for replacing its specialized parallel prefix adder, with another adder architecture meeting particular design measures in time, area, or power consumption. Neither have we encountered, in the open literature, any modulo 2^n+1 generic addition scheme with a single abstract n-bit adder, replaceable by any desired functionally equivalent concrete n-bit adder.

The rest of this paper is organized as follows. We review the conventional number representations for modulo 2^n+1 numbers and the related adder designs in Section 2. In Section 3, we propose a one-step generic modulo 2^n+1 addition scheme based on double-lsb [9] representation of modulo 2^n+1 residues [10]. Conversion to/from binary is discussed in Section 4, and finally we draw our conclusions in Section 5.

2. General Module 2^n+1 Addition

A general block diagram, for modulo 2^n+1 adders, is depicted in Figure 1. The main function is done by the middle block, namely an abstract n-bit-long adder, where an n-bit-long operation is formally defined, for the purpose of reference in this paper, as:

**Definition 1** (n-bit-long operation): An n-bit unary or binary operation, performing on one or two n-bit operands and producing the result with an inconstant latency with respect to n, but at most linearly depending on n, is called an n-bit-long operation.

Also we define a one-step modulo 2^n+1 adder as:

**Definition 2** (One-step modulo 2^n+1 adder): Any modulo 2^n+1 adder, whose critical delay path passes through only one n-bit-long binary operation (see Def. 1) is called a one-step modulo 2^n+1 adder.

If the top and bottom blocks of Figure 1 perform in constant time the adder is, by Def. 2, a one-step adder.

![General block-diagram of modulo 2^n+1 adder](image)

Figure 1. General block-diagram of modulo 2^n+1 adder

Examples of n-bit-long operations include:

- n-bit ripple-carry adders
- n-bit carry-accelerate adders (e.g., carry look-ahead [2] including parallel prefix [7], carry-select [8], or carry-skip [11] adders)
- n-bit increment/decrement adders
- n-bit zero detectors

The preprocessing and postprocessing blocks of Figure 1, depending on the underlined algorithm used for implementing modulo 2^n+1 adders, may vary in performance and complexity. For example, a straight forward algorithm may be described as follows:

**Algorithm 1 (Modulo 2^n + 1 Addition):**

**Inputs:** A = a_n a_{n-1} ... a_1 a_0, B = b_n b_{n-1} ... b_1 b_0 (0 ≤ A, B ≤ 2^n).

**Output:** S = (A + B)_{2^n+1} (0 ≤ S ≤ 2^n).

1. Compute \( W = A + B \) (\( W = w_{n+1} w_n W' \), \( W' = w_{n+1} \cdots w_1 w_0, w_{n+1} = a_n b_n \)).
2. If \( w_{n+1} = 1 \) then \( S = 2^n - 1 \)
   else if \( w_n = 0 \) then \( S = W' \)
   else \( S = W' - 1 \).

Step II of Algorithm 1 is justified as:

- \( w_{n+1} = 1 \) (i.e., \( A + B = 2^n \Rightarrow a_n = b_n = 1 \)):
  \( S = (A + B)_{2^n+1} = A + B - (2^n + 1) = 2^n - 1 \)
- \( w_{n+1} = w_n = 0 \) (i.e., \( A + B < 2^n \)):
  \( S = (A + B)_{2^n+1} = W' \)
- \( w_{n+1} = 0 \) and \( w_n = 1 \) (i.e., \( 2^n ≤ A + B < 2^{n+1} \)):
  \( W' = 0 \Rightarrow A + B = 2^n \), otherwise
  \( S = (A + B)_{2^n+1} = A + B - 1 - 2^n = W' - 1 \)
In the above algorithm, Step I involves an \( n \)-bit-long addition and Step II, in the worst case, goes through an \( n \)-bit-long decrement operation (i.e., to compute \( W' - 1 \)). Moreover checking for \( W' = 0 \), is generally an \( n \)-bit-long operation. It is naturally desirable to find algorithms that compute the required modulo addition by only one \( n \)-bit-long operation. Parallel computation of \( W \) and \( W - 1 \) is an alternative solution, but with high penalty in area and power consumption due to an extra active \( n \)-bit-long adder. As another solution use of diminished-1 representation has been proposed in [12] with further elaboration in [3]. Yet it requires either two \( n \)-bit-long addition operations in sequence or two \( n \)-bit-long adders that operate in parallel. For ease of reference, we provide Algorithms 2 for the latter method.

Algorithm 2 (Modulo 2\(^{n+1}\) Diminished-1 Addition):

Inputs: Diminished-1 representations of \( A \) and \( B \) (i.e., \( (z_a, A') \) and \( (z_b, B') \) with \( A' = A - 1, B' = B - 1 \), \( z_a = z_b = 1 \) for \( A, B > 0 \), and \( z_a = z_b = 0 \), for \( A = B = 0 \), where \( A' = a_{n-1} \ldots a_1 a_0 \) and \( B' = b_{n-1} \ldots b_1 b_0 \).

Output: Diminished-1 representation of \( S = (A + B) \) 2\(^{n+1}\) (i.e., \( (z_s, S) \), with \( 0 \leq S' = (A' + B' + 1)_{2^n} \leq 2^{n-1} - 1 \) and \( z_s = 1 \) for \( S > 0 \), and \( z_s = 0 \) for \( S = 0 \)).

I. Derive \( z_s = z_a \ OR \ z_b \).
II. If \( z_s = 1 \) then compute \( W = A' + B' + 1 \) (\( W = w_n w_{n-1} \ldots w_1 w_0 \)) else exit (\( S = 0 \)).
III. If \( w_n = 1 \) then \( S' = w_{n-1} \ldots w_1 w_0 + 1 \) else \( S' = W \).
IV. If \( S' = 0 \) then \( z_s = 0 \). \( \blacktriangleleft \)

Step II of Algorithm 2 is justified as:
- \( z_s = 1; S' = S - 1 = A + B - 1 = A' + 1 + B' + 1 - 1 = A' + B' + 1 \)
- \( z_s = 0; z_a = z_b = 0 \Rightarrow A = B = 0 \Rightarrow S = 0 \)

Step III is justified as:
- \( z_s = w_n = 1; W \geq 2^n \Rightarrow S' = (A' + B' + 1)_{2^n} = (A' + B')_{2^n} = (W + 1)_{2^n} = w_{n-1} \ldots w_1 w_0 + 1 \)
- \( z_s = 1, w_n = 0; W \leq 2^n \Rightarrow S' = (A' + B' + 1)_{2^n} = A' + B' + 1 = W = w_{n-1} \ldots w_0 \)

There is an \( n \)-bit-long addition operation, an \( n \)-bit-long increment, and an \( n \)-bit-long zero detection in Algorithm 2 (see Step II, III, and IV, respectively). The increment operation has been successfully fused in Step II through a special parallel prefix addition scheme [6]. As a further interesting innovation [7] provides a novel modulo 2\(^{n+1}\) addition scheme (described, in our notation, by Algorithm 3, below), based on Equation (2), where \( W = A + B + 2^n - 1 \) and no special provision for detection of zero results is needed.

\[
S = (A + B)_{2^{n+1}} = \begin{cases} (W)_{2^{n+1}} & \text{if } W \geq 2^{n+1} \\ ((W_{2^n} + 2^n + 1)_{2^{n+1}} & \text{otherwise} \end{cases} \tag{2}
\]

Algorithm 3 (Modulo 2\(^{n+1}\) Parallel-Prefix Addition):

Inputs: \( A = a_n a_{n-1} \ldots a_1 a_0, B = b_n b_{n-1} \ldots b_1 b_0 \) (\( 0 \leq A, B \leq 2^n \)).
Output: \( S = (A + B)_{2^{n+1}} (0 \leq S \leq 2^n) \).

I. Compute \( W = A + B + 2^n - 1 \) (\( W = w_n w_{n-1} \ldots w_1 w_0 \)) by:
   a. Simplified carry-save addition of \( A, B, \) and \( 2^n - 1 \).
   b. Parallel prefix addition of carry and sum vectors resulted from a.

II. \( S = s_n s_{n-1} \ldots s_1 s_0 = W_{n+1} + w_n \ldots w_1 w_0 + w_{n+1} \).

The function of Algorithm 3 is illustrated in Figure 2, and Step II is justified as:
- \( w_{n+1} = 1; 2^{n+1} \leq W \leq 2^{n+1} + 2^n - 1 \Rightarrow w_n = 0, \) and \( A + B + 2^n - 1 \geq 2^{n+1} \Rightarrow S = (A + B)_{2^{n+1}} = A + B - 2^n - 1 = W - 2^{n+1} = 0 \).
- \( w_{n+1} = 0, w_n = 1; 2^n \leq W < 2^{n+1} \Rightarrow 1 \leq A + B < 2^n + 1 \Rightarrow S = (A + B)_{2^{n+1}} = A + B = W - 2^n + 1 = 0 \).

\( W_n = w_n \ldots w_1 w_0 + 1 \).

The cases of \( A + B = 2^n + 1, \) and \( A = B = 0, \) lead to \( W = 2^{n+1} \) and \( W + 2^n + 1 = 2^n + 1 = 1 + 2^n + 1 = 2^{n+1} \), respectively. Both cases result in \( S = 0 \). The \( n \)-bit-long addition of Step 2b can be implemented by a variety of carry acceleration methods [11]. In particular carry look-ahead adders [2] and parallel prefix adders [7] can be used to fuse the final increment into positional carry computation of the main addition for derivation of \( W \). The trick, as shown in

![Figure 2. Derivation of W in Algorithm 3](image-url)
Expression (3) is to postpone the computation involving \( c_m \) (i.e., carry into the lsb position) to the last stage of positional carry derivations, thus avoiding a new carry look-ahead computation for enforcing the end-around carry.

\[
c_i = G_{i+1} + \overline{P_{i+1} c_m} = G_{i+1}, \quad c_{i+1} = G_i, \tag{3}
\]

\( c_i \) is the carry into position \( i \) (\( c_0 = c_m = 0 \)), \( G \) and \( P \) variables represent \( c_m \)-independent generate and propagate expressions based on positional generate and propagate signals (i.e., \( g_i \) and \( p_i \)), and \( c^*_i \) is the final carry into position \( i \) used for derivation of \( s_i = p_i \oplus c^*_i \). Expressions for \( G_i \) and \( P_i \) variables (including \( G_n \)) depend on all the \( g_j \) and \( p_{ji} \) signals \((0 \leq j \leq i)\). These are computable in parallel by fundamental carry look-ahead operator cells [13] or parallel prefix trees through the recurrence equation-set (4).

\[
G_i = g_{i+1} + p_{i+1} G_{i+2}, \quad P_i = p_{i+1} P_{i+2}, \tag{4}
\]

\[
g_i = a_i \oplus b_i (a_{i+1} + b_{i+1}), \quad p_i = a_i \oplus b_i + (a_{i+1} + b_{i+1})
\]

Direct computations of \( s_i = p_i \oplus c^*_i \) and \( s_n = p_n \oplus c^*_n \) obviates the need for computations of \( w_i = p_i \oplus c_i \) and \( s_n = w_{n+1} + w_n \). The fused end-around carry is \( w_{n+1} = G_n + a_n b_n \), where \( G_n \) and \( a_n b_n \) cannot both be equal to 1. Therefore the overall delay for sum bits \( s_n \), based on the same analysis as in [7], consists of the following delay components, leading to \((7 + 2 \lceil \log n \rceil)\) unit gate delays:

a. The XOR operation in the original carry-save addition: 2 unit gates
b. \( g_i \) and \( p_i \), computations: 1 unit gate
c. \( G \) and \( P \), derivations: 2 log \( n \) unit gates (Parallel prefix logic assumed)
d. \( c^*_i \) computation: 2 unit gates
e. The XOR operation for the final sum generation: 2 unit gates

The overall latency of the similar implementation in [7] is estimated to be \((9 + 2 \lceil \log n \rceil)\) unit gate delays. Our better estimate is due to using a modified equation for \( c^*_i \) computation (see equation (3) above), which unfortunately has the disadvantage of large fan-out on \( a_n b_n \).

The best previous result for fastest modulo \( 2^n + 1 \) addition, as noted in Section 2 above, has been reported in [7] with the latency of \((6 + 2 \lceil \log n \rceil)\) unit gates, where the proposed architecture (see Figure 3, copied from [7], below) is based on a complex parallel prefix tree with the following VLSI-unfriendly properties:

- Double fundamental carry look-ahead operator cells in some tree nodes
- Different fundamental carry look-ahead operator cells in the last row (pale circles)

Another disadvantage is the non-generic behavior of the design in Figure 3, where it is not possible to replace the main \( n \)-bit-long adder by other adder architectures (e.g., when due to economizing area and power consumption, designers prefer to do Step I.b by a ripple-carry adder).

![Figure 3. Modulo \( 2^n + 1 \) parallel prefix adder with \( 2^n \) backward lines, from [7]](image)

All the previously published algorithms for modulo \( 2^n + 1 \) addition, including the ones described in our notation by Algorithms 2 and 3 above, essentially require an \( n \)-bit-long addition operation followed by an \( n \)-bit-long increment operation, unless special carry acceleration technique offered in [7] or parallel adders to compute \( sum \) and incremented \( sum \) are used. In the next section, however, we present our novel modulo \( 2^n + 1 \) representation, which is suitable for one-step ripple-carry (or any carry-accelerate) implementation of modulo \( 2^n + 1 \) adders.

### 3. Double-LSB Modulo \( 2^n + 1 \) Addition

The following new one-step modulo \( 2^n + 1 \) addition algorithm is based on double-LSB [9 and 14] representation of modulo \( 2^n + 1 \) numbers [10]. An \( n \)-binary-position double-LSB representation is basically the same as an \( n \)-bit unsigned binary representation, except for an extra bit in position zero. The extra \( lsb \) enhances the range of represented numbers to exactly \([0, 2^n]\) and leads to the possibility of storing the end-around carry instead of a post-increment operation. The two equally weighted \( lsb \)s are distinguished by primed and non-primed versions of the same variable.
Algorithm 4 (Addition of Double-lsb Modulo 2^n + 1 numbers):

Inputs: A = a_{n-1}...a_1a_0', B = b_{n-1}...b_1b_0' (0 \leq A, B \leq 2^n).

Output: S = (A + B)^{2^n+1} = s_{n-1}...s_1s_0s_0' (0 \leq S \leq 2^n).

I. Compute W = A + B - 1.
II. If W \geq 2^n then S = W - 2^n (s_0' = 0) else S = W + 1 (s_0' = 1).

The above algorithm, in an abstract sense, is the same as Algorithm 3 of the previous section. But the double-lsb representation of the operands and the following implementation of A + B - 1 exhibit a considerably different implementation approach. The basic idea, as depicted in Figure 4 is to compute W as follows:

- If \( z = a_0 + a_0' + b_0 + b_0' = 1 \) then \( W = A + B - z \).
- If \( a_0 + a_0' + b_0 + b_0' = 1 \) (i.e., \( z = 0 \) and at least one of the lsb's is 1) then we ignore a 1-valued lsb, to account for \(-1\) in Step I above, and repeat the sum of other three bits as \( t_1u_0 \) in Figure 4. Therefore \( W = A + B - z - 1 \).

The latter observation suggests the computing of \( A + B - z \), in both cases of \( z = 1 \) or \( z = 0 \), with the understanding that in the second case (i.e., \( z = 0 \)), the decrement operation is taken care of by leaving out a 1-valued lsb. In Figure 4, \(-z\) is represented as \(-2^{n+1}z(2^{n+1} - 1)\). Table 1 shows how a 1-valued bit is left out, where sum of the three remained bits are derived as \( 2t_1 + u_0 \) (Equation (5)) and due to \( a_0 = a_0' = b_0 = b_0' = 0 \), for \( z = 1 \), three entries are missing.

\begin{align*}
  &t_1 = a_0b_0a_0' + a_0b_0b_0' + a_0'b_0a_0 + a_0'b_0b_0' \quad \text{(5)} \\
u_0 = a_0 \oplus b_0 \oplus a_0' \oplus b_0' \\
0 &\quad a_0 \quad \ldots \quad \ldots \quad a_1 \quad a_0 \\
0 &\quad b_0 \quad \ldots \quad \ldots \quad b_1 \quad b_0 \\
-z &\quad z \quad \ldots \quad \ldots \quad z \quad z \\
-z &\quad a_0 \oplus b_0 \oplus z \quad \ldots \quad \ldots \quad a_1 \oplus b_1 \oplus z \quad u_0 \\
t_1 &\quad t_{n-1} \quad \ldots \quad t_2 \quad t_1 \\
c_{n+1} &\quad w_n \quad \ldots \quad w_2 \quad w_1 \quad w_0 \\
&\quad s_{n-1} \quad \ldots \quad s_2 \quad s_1 \quad s_0 \\
s_0' &\quad w_n \quad \ldots \quad w_2 \quad w_1 \quad w_0 
\end{align*}

Table 1. Derivation of three remained bits in position 0

There is actually no need to compute \( c_{n+1} = z \), in position \( n+1 \) of Figure 4, for \( c_{n+1} \) and \( z \) may directly contribute to the derivation of the final sum \( S \) as is outlined in Table 2 and Equation-set (6), where the absence of three other potential entries are due to the constraint \( 0 \leq A + B - 1 < 2^{n+1} \).

\begin{align*}
s_i = w_j (c_{n+1} + z^i), &\quad \text{for } 0 \leq i \leq n, \quad s_0' = w_n \\
&\quad \text{(6)}
\end{align*}

A parallel prefix implementation of Algorithm 4 is depicted in Figure 4, where the delay components as listed below sum up to the same latency as the totally parallel prefix (TPP) implementation of Algorithm 3 in [7] (i.e., \( 6 + 2 \left\lceil \log n \right\rceil \)), assuming equal delay for full adders in Figure 5 and modified half-adders of Figure 3):

a. Computation of \( t_1 \) (Equation (5)), and delay of the CSA adder: 2 unit gates.

b. \( g_i \) and \( p_i \) computations: 1 unit gate

c. \( G \) and \( P \) derivations: 2 \( \log n \) unit gates (Parallel prefix logic assumed)

d. The XOR operation for the interim sum generation: 2 unit gates

e. The AND delay for final sum generation: 1 unit gate

However our parallel prefix tree is without global backward lines, and enhanced VLSI regularity is clearly an additional advantage in our design.

### 4. Conversion to/from Binary

Conversion of a binary integer \( I \) to double-lsb modulo \( 2^n+1 \) representation, follows the conventional process of binary to modulo \( 2^n+1 \) residue conversion up to reducing \( I \) to an \( n+1 \)-bit binary integer [11]. Then the process is continued, by subtracting the msb (which weighs \( 2^n \)) from the rest of representation (i.e., the rightmost \( n \) bits), and saving the outgoing borrow as the second lsb. The latter is 1 if and only if
the original $n+1$-bit number had been equal to $2^n$. The second part of the conversion process is less complex than conventional binary to modulo $2^n+1$ conversion which, besides subtraction of msb, involves an $n$-bit zero detection for the case of $2^n$ as the residue.

For the reverse conversion, the second lsb does not introduce considerable inefficiency. Residue to binary converters, normally implement the Chinese remainder theorem [1] using carry save adders (CSA) for the required multi-operand binary addition. With a careful design of the CSA tree, contribution of the second lsb to the overall conversion delay may only add one level of CSA. For example, for the popular moduli set $\{2^n+1, 2^n\}$, Equation-set (11) adapted from [15] converts $(X, Y, Z)$ to $I$, where $X = (I)_{2^n}^-, Y = (I)_{2^n-1}$, and $Z = (I)_{2^n+1}$:

$$I = 2^n I + X I = (2^n X + Z) + 2^{n-1} (2^n + 1)(Y + Z) Z_{2^n} \tag{11}$$

The equation for $I'$, after some manipulation, turns to Equation (12), where $\overline{X} (\overline{Z})$ is one’s complement of $X (Z)$.

$$I' = 2^n \overline{X} + Z + 2^{n-1} (Y + Z) + 2^{n-1} (Y + Z) Z_{2^n-1} \tag{12}$$

Assuming $X = x_0, \ldots, x_0$, $Y = y_0, \ldots, y_0$, and $Z = z_{n-1}, \ldots, z_0$, implementation of (12) may be illustrated as in Figure 6. It is easily seen that the second lsb (i.e., $z'_0$) deepens the multi-operand addition tree by only one level. Therefore, a $(4; 2)$ compressor may be used for reduction of each column instead of a full adder. The additional delay is equal to that of one XOR gate. This is practically negligible in comparison with the long $2n$-bit addition required after reduction.

Figure 6. Multi operand addition tree

5. Conclusion

We used double-lsb representation for modulo $2^n+1$ numbers, and presented a modulo $2^n+1$ addition scheme, where the end-around carry may be stored as the second lsb instead of an undesired $n$-bit-long increment operation. With the double-lsb representation, our algorithm is the first, in the open literature, that allows for a one-step $n$-bit carry-ripple implementation of modulo $2^n+1$ addition. However, there is a preprocessing phase consisting of a constant-time carry-save addition.

This design is a generic one in a sense that the main $n$-bit-long adder may be replaced by any adder architecture that fits the designer’s goals for area, time, and power requirements. The carry-save and carry-ripple adder
combination, leads to less area consumption than that of the parallel addition paradigm with n-bit selector logic. Moreover, less power dissipation is expected due to less switching activity of carry-save adder. Furthermore we showed that a conventional parallel prefix implementation of the double-LSB addition algorithm leads to the same latency as the best reported delay-optimized modulo 2^n+1 adder [7]. Nevertheless, our design is more VLSI-friendly with respect to regularity and lack of backward interconnection lines, except for one due to storing the end-around carry. Finally we showed that double-LSB representation does not introduce any inefficiency for binary to residue conversion and causes negligible additional delay in the reverse conversion for the moduli set \{2^n-1, 2^n, 2^n+1\}. Given that efficient modulo 2^n + 1 multipliers have been designed (e.g., [16]), further research is ongoing for modulo 2^n + 1 multiplier design based on double-LSB representation, and similar unconventional representations for other moduli are being explored.

Acknowledgement

This research was funded by Shahid Beheshti University under contract # 600/1720/1101.

References


Ghassem Jaberipur received his B.S in Electrical Engineering and PhD in Computer Engineering from Sharif University of Technology in 1974 and 2004, respectively, his M.S in Engineering (majoring in computer hardware) from UCLA in 1976, and his M.S in Computer Science from University of Wisconsin in Madison in 1979. Since 1979, he has been with the department of Electrical and Computer Engineering of Shahid Beheshti University (Tehran, Iran). His main activities include teaching undergraduate courses in theory and implementation of programming languages, and graduate courses in compiler construction and computer arithmetic. His main research interest is in computer arithmetic. Dr. Jaberipur is also affiliated with the School of Computer Science, IPM (Tehran, Iran).

E-mail: Jaberipur@SBU.ic.ir