Received April 9, 2019, accepted April 26, 2019, date of publication May 3, 2019, date of current version May 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2914641

Fast Large Integer Modular Addition in GF(p) Using Novel Attribute-Based Representation

BADER ALHAZMI[1,2], **(Member, IEEE), AND FAYEZ GEBALI**[1], **(Life Senior Member, IEEE)**
[1]Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC V8W 3P6, Canada
[2]Department of Computer Engineering, Umm Al-Qura University, Makkah 21955, Saudi Arabia

Corresponding author: Fayez Gebali (fayez@uvic.ca)

This work was supported by the Ministry of Education of Saudi Arabia and Umm Al-Qura University.

ABSTRACT Addition is an essential operation in all cryptographic algorithms. Higher levels of security require larger key sizes and this becomes a limiting factor in $GF(p)$ using large integers because of the carry propagation problem. We propose a novel and efficient attribute-based large integer representation scheme suitable for large integers commonly used in cryptography such as the five NIST primes and the Pierpont primes used in supersingular isogeny Diffie–Hellman (SIDH) for post-quantum cryptography. Algorithms are proposed for this new representation to implement arithmetic operations such as two's complement, addition/subtraction, comparison, sign detection, and modular reduction. Algorithms are also developed for converting binary numbers to attribute representation and vice versa. The extensive numerical simulations were done to verify the performance of the new number representation. Results show that addition is done faster in our proposed representation when compared with binary and residue number system (RNS)-based additions. Attribute addition outperformed RNS addition for all values of m where $128 \leq m \leq 32\,768$ bits for all machine word sizes w where $4 \leq w \leq 128$ bits. Attribute-based addition outperforms Kogge–Stone binary adders for a wide range of m when w is small. For increasing values of w, the speed advantages are evident only for large values of m. This makes the proposed number representation suitable for implementing cryptographic applications in embedded processors for IoT and consumer electronic devices where w is small.

INDEX TERMS Prime fields $GF(p)$, large integer arithmetic, modular arithmetic, Kogge-Stone adder, number representation, post-quantum cryptography, SIDH, cryptographic processor, embedded systems, NIST primes, generalized pierpont prime, parallel algorithms.

I. INTRODUCTION

Large integers are used in several key areas such as cryptographic systems, digital signal processing, and fault-tolerant applications. Large integer representation has direct impact on the efficiency of the calculations in hardware/software implementations. However, operations on large integers suffer from the long carry propagation delays. Residue number system (RNS) is a commonly used representation to solve the carry propagation problem. However, RNS has several limitations such as conversion from/to binary representation and difficulty in determining basic properties such as number magnitude, sign, overflow or ability to compare two numbers.

For these reasons, most cryptographic systems adopted the polynomial field $GF(2^m)$, instead of the integer field $GF(p)$ because of the absence of carry propagation. Recently these cryptographic systems proved vulnerable to side-channel and quantum computing attacks. To meet these challenges, authors are exploring cryptographic protocols that use large primes in $GF(p)$. National institute of standards and technology (NIST) has selected five large primes, which are generalized Mersenne numbers [1]. These primes were selected with an eye to simplify the modular reduction operations for machine word sizes that were either 32 or 64 bits. The authors in [2] proposed a post-quantum cryptographic technique that uses generalized Pierpont primes and Supersingular Isogeny Diffie-Hellman (SIDH) key exchange protocol to provide immunity to quantum attacks. Efficient algorithms tailored for efficient modular arithmetic are yet to be developed, apart from using the RNS approach.

The new number representation proposed here does not have the disadvantages of the RNS number representation,

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

as will be discussed in detail in Sec. II. The main contributions of this work are:

1) Propose a new non-positional attribute-based large integer representation.
2) Develop algorithms for converting a binary number to attribute representation, and vice versa.
3) Develop algorithms to perform several arithmetic operations based on the new representation.
4) Perform numerical simulations to verify the advantages of the new representation.

This paper is organized as follows. In Sec. II we provide a review of related works dealing with arithmetic operations using large integers. In Sec. III we present the proposed attribute-based large integer representation. In Sec. IV we develop algorithms based on our new representation to perform the basic arithmetic operations such as addition/subtraction, two's complement, comparison sign detection and modular addition/subtraction. In Sec. VII we provide numerical simulations and software implementations to verify the performance of the proposed representation. In Sec. VIII we summarize the main conclusions of this paper.

II. RELATED WORKS

The choice of integer representation affects the performance of the basic arithmetic operations, especially when dealing with large integers. The delay of arithmetic operations with large integers is extremely long due to carry propagation. The Residue Number System (RNS) [3], a non-positional number representation, has been proposed to overcome the carry delay problem. RNS allows representing large integers as a set of smaller integers to achieve fast and parallel arithmetic operations for addition, subtraction, and multiplication since they are performed on shorter operands [4]–[6]. This property has attracted the attention of many researchers to utilize RNS in many applications in digital signal processing systems [7], [8], error detection and correction and fault-tolerant applications [9], embedded systems [10], and asymmetric cryptography systems [11]–[18].

However, RNS suffers from several serious drawbacks:

1) It is difficult and/or slow to convert data between the RNS and their binary equivalents [19]–[22].
2) The sign of the data is not easily determined [23]–[26].
3) It is not easy to compare two numbers in RNS domain to determine equality or inequality [27]–[31].
4) It is hard to detect an overflow that might happen as a result of an operation [32], [33].
5) It is necessary to perform the expensive conversion to binary representation after each arithmetic operation to be able to extract the state of the arithmetic results.
6) Division by a constant is difficult to implement with RNS representation [34]–[37].
7) It is inefficient to perform the division operation with RNS representation due to the combination of iterated subtractions and comparisons operations [38]–[43].

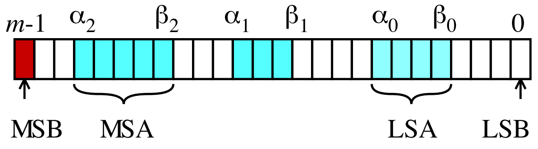


FIGURE 1. An integer with three strings of ones. MSB: Most-significant bit, LSB: Least-significant bit, MSA: Most-significant attribute, and LSA: Least-significant attribute.

As will be seen in the sequel, the attribute-based large integer number representation proposed here does not suffer from these disadvantages. This will lead to dramatic improvements in the system performance, as will be proven in Sec. VII.

Binary addition algorithms are used in most, if not all, low-power embedded processors as well as high-performance servers. This is due to the simplicity of adding binary numbers. However, the carry propagation problem plagues binary addition and is the main factor that determines the speed of operation of the processor ALU. The binary ripple carry adder (RCA) is the simplest and slowest type of binary adders. However, it is widely used and still serves as the basis for comparing the performance of other addition algorithms.

There are many types of fast binary adders that have been proposed in the literature such as carry skip adders (CSK) [44], [45], carry select adders (CSL) [46], [47], carry save adders (CSA) [48], carry lookahead adders (CLA) [49], [50], and parallel prefix adders (PPF) [51]–[56]. Many variations and combinations on these basic binary adders have also been proposed such as the hybrid carry lookahead/carry select adder [57], hybrid ripple carry/hierarchical carry lookahead type 2 adder [58] and hybrid parallel prefix/carry select and skip adder [59].

Most of the proposed works in the area of PPF adders focus on improving the performance of the hardware implementation in terms of delay, area, and power. Other works reported combinations of PPF adders and basic binary adders. A comparative analysis of PPF adders are given in [60]–[63].

III. PROPOSED ATTRIBUTE-BASED INTEGER REPRESENTATION

An m-bit two's complement integer has the binary representation:

$$N = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \tag{1}$$

Figure 1 shows the m-bit representation where the red box indicates the sign bit, the blue boxes indicate the non-zero bits, and the white boxes indicate the zero bits.

Referring to Fig. 1 and (1), we can represent N in terms of the non-zero values of a_i as:

$$N = \sum_{i=\beta_2}^{\alpha_2} 2^i + \sum_{i=\beta_1}^{\alpha_1} 2^i + \sum_{i=\beta_0}^{\alpha_0} 2^i \tag{2}$$

In general when we have L contiguous strings of 1's, the above equation becomes:

$$N = \sum_{i=\beta_{L-1}}^{\alpha_{L-1}} 2^i + \sum_{i=\beta_{L-2}}^{\alpha_{L-2}} 2^i + \cdots + \sum_{i=\beta_0}^{\alpha_0} 2^i \qquad (3)$$

Our proposed integer representation is based on the above equation. We use a short-hand notation to represent and to store in memory the number in terms of the summation limits indicated in (3):

$$N \equiv \{(\alpha_{L-1},\ \beta_{L-1}), (\alpha_{L-2},\ \beta_{L-2}), \ldots, (\alpha_0,\ \beta_0)\} \qquad (4)$$

We call the tuple or pair (α_i, β_i) the i-th attribute of the number.

Equation (4) indicates that the number N can be represented by the set of (α,β) attributes. This list N can be stored as an abstract data type single- or doubly-linked lists [64].

In (4), the attribute $(\alpha_0,\ \beta_0)$ is called the least-significant attribute (LSA), as shown in Fig. 1. Similarly, the attribute $(\alpha_{L-1},\ \beta_{L-1})$ is called the most-significant attribute (MSA), as shown in Fig. 1.

Assuming our integers are represented using m-bits, each α or β is an integer value that would require a bits where:

$$a = \lceil \log_2\ m \rceil \qquad (5)$$

The following lemma proves the relationship between the values of α and β attributes of a number.

Lemma 1: For a given number, the values of α and β must satisfy the following inequalities

$$\alpha_i \geq \beta_i \qquad (6)$$
$$\beta_{i+1} > \alpha_i + 1, \quad 0 \leq i < L \qquad (7)$$

Proof: From (3) the least value for the upper limit of each summation is when $\alpha_i = \beta_i$. Hence we have in general $\alpha_i \geq \beta_i$. This proves (6).

Since, there is at least one bit gap to separate any contiguous string of ones, the value of β_{i+1} cannot equal the value of α_i. This proves (7). ■

The following lemma gives an upper limit to the maximum number of attributes of an integer.

Lemma 2: The maximum number of attributes of an integer is $m/2$.

Proof: Assume all the attributes have the same length l_a:

$$l_a = \alpha - \beta + 1$$

Assume also that the number of zeros between attributes (l_0) is equal. The number of attributes will be given by:

$$n_a = \frac{m}{l_a + l_0}$$

The maximum number of attributes is when l_a and l_0 are at their least possible values.

From Lemma 1, the least value for $l_a = 1$ and the least value of $l_0 = 1$. Hence maximum number of attributes is given by:

$$n_a = \frac{m}{2}$$

■

The following lemma shows how the sign of an integer number can be inferred from its attributes representation.

Lemma 3: Given an m-bit integer with L attributes, the sign of that integer can be inferred from the value of α_{L-1}.

Proof: From (3), α_{L-1} represents the position of the most significant 1 in the number N. When $\alpha_{L-1} = m - 1$ we have a negative number since the sign bit at location $m - 1$ is 1, according to (1). Conversely, when $\alpha_{L-1} < m - 1$ we have a positive number since the sign bit at location $m - 1$ is zero. ■

The following lemma proves how to infer if a number is even or odd based on its attributes.

Lemma 4: Given an m-bit integer with L attributes, the number is even or odd from value of β_0.

Proof: From (3), β_0 represents the position of the least significant 1 in the number N. When $\beta_0 = 0$ we have an odd number since the bit at location 0 is 1, according to (1). Conversely, when $\beta_0 > 0$ we have an even number since the bit at location 0 is zero. ■

A. ATTRIBUTE REPRESENTATION OF NIST PRIMES

We illustrate in this section how the NIST primes are expressed using the proposed attribute-based representation. NIST proposed five primes for elliptic curve cryptography [1]:

$$\text{P-192} = 2^{192} - 2^{64} - 1 \qquad (8)$$
$$\text{P-224} = 2^{224} - 2^{96} + 1 \qquad (9)$$
$$\text{P-256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \qquad (10)$$
$$\text{P-384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \qquad (11)$$
$$\text{P-521} = 2^{521} - 1 \qquad (12)$$

The binary representations of these five primes are given by:

$$\text{P-192} = \sum_{i=65}^{191} 2^i + \sum_{i=0}^{63} 2^i \qquad (13)$$
$$\text{P-224} = \sum_{i=96}^{223} 2^i + \sum_{i=0}^{0} 2^i \qquad (14)$$
$$\text{P-256} = \sum_{i=224}^{255} 2^i + \sum_{i=192}^{192} 2^i + \sum_{i=0}^{95} 2^i \qquad (15)$$
$$\text{P-384} = \sum_{i=129}^{383} 2^i + \sum_{i=96}^{127} 2^i + \sum_{i=0}^{31} 2^i \qquad (16)$$
$$\text{P-521} = \sum_{i=0}^{520} 2^i \qquad (17)$$

The attribute-based representations of five NIST primes are given by:

$$\text{P-192} \equiv \{(191, 65), (63, 0)\} \qquad (18)$$
$$\text{P-224} \equiv \{(223, 96), (0, 0)\} \qquad (19)$$
$$\text{P-256} \equiv \{(255, 224), (192, 192), (95, 0)\} \qquad (20)$$
$$\text{P-384} \equiv \{(383, 129), (127, 96), (31, 0)\} \qquad (21)$$
$$\text{P-521} \equiv \{(520, 0)\} \qquad (22)$$

From the above equations, it becomes obvious that attribute-based NIST primes representation is very concise and requires a small number of entries compared to storing all m bits of the prime.

B. REPRESENTATION OF THE NUMBER 0

We need to consider how to represent the integer 0 when our number is stored as a linked list in (4). For the case of a linked list representation, we assign the start address the value `NULL`.

C. CONVERSION FROM BINARY TO ATTRIBUTE REPRESENTATION

Unlike other number representations, converting from binary to attribute representation is a simple process. Converting a binary number to attribute representation requires a simple scanning, starting from the LSB to the MSB, or vice versa, for contiguous strings of ones. The length of each string could vary between 1 to m. For each string the position of the starting 1 is assigned to the β value of that string. The end position of the last 1 is assigned to the α value of that string.

However, the scanning process requires long time to complete especially for large numbers. One option divides the m bits into multiple zones where each zone is of length R bits. Conversion from binary to attribute proceeds in parallel in each zone.

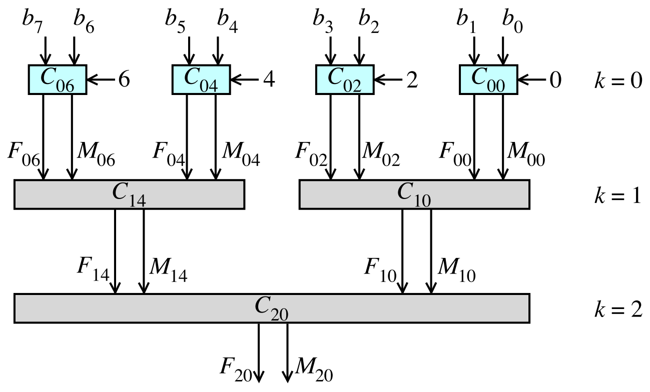
The binary to attribute conversion in each zone can be performed in a binary tree with $\mathcal{O}(\log_2\ R)$ complexity. Algorithm 1 shows the pseudo code for conversion from an R-bit binary number to its attribute representation. The algorithm requires k iterations where $k = \log_2 R$. In the first iteration where $k = 0$ (Lines 3 – 22), each pair of binary inputs are converted to a single attribute and the flag F will be set to one indicating that there is an attribute generated from the conversion. Otherwise, the value of the flag F will be set to zero. The maximum length L is one in this level of conversion. Next iterations where $k > 0$ (Lines 23 – 51), the algorithm will iterate to combine any two attributes that are separated with one value (α in next attribute is equal β of first attribute + 1) and generate the required flag, set the length, and update M for next iterations. Figure 2 shows an example for conversion from binary to attribute representation for R-bit word size when $R = 8$.

D. CONVERSION FROM ATTRIBUTE REPRESENTATION TO BINARY

Unlike other number representations, converting from attributed representation to binary is a simple process. Converting attributes of a number to its binary equivalent requires building a m-bit string of zeros. For each (α, β) pair, a string of ones is inserted in the binary number starting at position β and ending at position α. This process can be done in parallel since there is no overlap between attributes positions. Algorithm 2 shows the pseudo code for conversion from attribute representation to R-bit binary number.

Algorithm 1 Pseudo Code for Conversion From an R-Bit Binary Number to Its Attributes Representation (BI_2_ATT)

```
Input: $N, R$
Output: $M, L, F$
 1: $k \leftarrow 0$                                        ▷ First Level
 2: {                                  ▷ Start of parallel code section
 3: for $i = 0 : 2 : R-2$ do
 4:   if $b(i) = 1$ then
 5:     $\beta(k,i) \leftarrow i; F(k,i) \leftarrow 1; L(k,i) \leftarrow 1$
 6:     if $b(i+1) = 1$ then
 7:       $\alpha(k,i) \leftarrow i+1$
 8:     else
 9:       $\alpha(k,i) \leftarrow i$
10:     end if
11:   else
12:     if $b(i+1) = 1$ then
13:       $\beta(k,i) \leftarrow i+1; \alpha(k,i) \leftarrow i+1$
14:       $F(k,i) \leftarrow 1, \quad L(k,i) \leftarrow 1$
15:     else
16:       $\beta(k,i) \leftarrow \{\}; \alpha(k,i) \leftarrow \{\}$
17:       $F(k,i) \leftarrow 0; L(k,i) \leftarrow 0$
18:     end if
19:     $M(k,i) \leftarrow \{(\alpha(k,i), \beta(k,i))\}$
20:   end if
21: end for
22: }                                    ▷ End of parallel code section
23: for $k = 1$ to $\log_2 R - 1$ do                          ▷ Next Levels
24:   {                                ▷ Start of parallel code section
25:   for $j = 0 : 2^{k+1} : R-1$ do
26:     if $F(k-1,j) = 0$ and $F(k-1,j+2) = 0$ then
27:       $M(k+1,j) \leftarrow \{\}$
28:       $F(k+1,i) \leftarrow 0; L(k+1,i) \leftarrow 0$
29:     else if $F(k-1,j) = 0$ and $F(k-1,j+2) = 1$ then
30:       $M(k+1,j) \leftarrow M(k,j+2)$
31:       $F(k+1,j) \leftarrow 1; L(k+1,j) \leftarrow 1$
32:     else if $F(k-1,j) = 1$ and $F(k-1,j+2) = 0$ then
33:       $M(k+1,j) \leftarrow M(k,j)$
34:       $F(k+1,j) \leftarrow 1; L(k+1,j) \leftarrow 1$
35:     else
36:       $X \leftarrow \text{end}(M(k,j)); Y \leftarrow \text{first}(M(k,j+2))$
37:       if $Y.\beta = X.\alpha + 1$ then
38:         $M_{\text{temp}} \leftarrow \{(X.\beta,\ Y.\alpha)\}$
39:         delete$(X, M(k,j))$
40:         delete$(Y, M(k,j+2))$
41:         $M(k+1,j) \leftarrow M(k,j+2) \parallel M_{\text{temp}} \parallel M(k,j)$
42:         $F(k+1,j) \leftarrow 1$
43:         $L(k+1,j) \leftarrow L(k,j) + L(k,j+2) - 1$
44:       else
45:         $M(k+1,j) \leftarrow M(k,j+2) \parallel M(k,j)$
46:         $F(k+1,j) \leftarrow 1$
47:         $L(k+1,j) \leftarrow L(k,j) + L(k,j+2)$
48:       end if
49:     end if
50:   end for }                          ▷ End of parallel code section
51: end for
52: return $M, L, F$
```



b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
C_{06} 6 C_{04} 4 C_{02} 2 C_{00} 0
$k=0$
F_{06} M_{06} F_{04} M_{04} F_{02} M_{02} F_{00} M_{00}
C_{14} C_{10}
$k=1$
F_{14} M_{14} F_{10} M_{10}
C_{20}
$k=2$
F_{20} M_{20}

FIGURE 2. **Conversion from binary to attribute representation for *R*-bit word size. Case when *R* = 8.**

Algorithm 2 Pseudo Code for Conversion From Attribute Representation to R-Bit Binary Number (ATT_2_BI)

Input: $N, L, m, R,$
Output: M

```
1: {                                  ▷ Start of parallel code section
2: $M(1 : m/R - 1) \leftarrow 0$
3: for $i = 0$ to $R - 1$ do
4:     for $j = 0$ to $L(i) - 1$ do
5:         $M(\beta(j) : \alpha(j)) \leftarrow 1$
6:     end for
7: end for
8: }                                  ▷ End of parallel code section
9: return $M$
```

IV. ARITHMETIC OPERATIONS USING ATTRIBUTES

Having proposed and defined the attributes for an integer, we are able now to propose algorithms for performing finite-field arithmetic operations using the attributes. There are several arithmetic operations that are needed. In this paper, we discuss the following operations:

1) Two's complement of a number: $-N$
2) Number addition/subtraction: $N_1 \pm N_2$
3) Number comparison: $N_1 ? N_2$
4) Modular addition/subtraction: $N_1 \pm N_2 \mod p$
5) Converting a binary number to attribute-based representation; and vice versa

A. ATTRIBUTE TWO'S COMPLEMENT ALGORITHM

In binary representation, two's complement is used to accommodate negative numbers. The most-significant bit is reserved as a sign bit. A positive number in has a zero sign bit, whereas a negative number has a one in the sign bit. There are two different ways to find the two's complement of a number. The first method first finds the one's complement then adding one. The second method scans for the first one from the least-significant bit then complements all the succeeding bits. In this paper, we will use the first method.

Algorithm 3 shows how to find one's complement for a number based on their attributes. The algorithm consists of three parts. The first part (Lines 1 – 4) to find the complement attribute before the first attribute in the number N. The second part (Lines 5 – 8) to find the complement for attributes in the number N. The third part (Lines 9 – 15) to find the complement attribute after the last attribute in the number N. Finally, the algorithm will return the one's complement N' for the input number N (Line 16.)

After finding the one's complement for the number, the next step will be adding one to the result of the first step N'. This step will be done by using Algorithm 4 as will be shown in Section IV-B.

Algorithm 3 Pseudo Code to Find One's Complement (ONESCOMP) for a Number Based on Their Attributes

Input: N, L, m
Output: N'

```
1: if $\beta_N(1) > 0$ then
2:     $\beta_{N'}(1) \leftarrow 0$
3:     $\alpha_{N'}(1) \leftarrow \beta_N(1) - 1$
4: end if
5: for $i = 2$ to $L - 1$ do
6:     $\beta_{N'}(i) \leftarrow \alpha_N(i-1) + 1$
7:     $\alpha_{N'}(i) \leftarrow \beta_N(i) - 1$
8: end for
9: if $\alpha_N(L) < m - 1$ then
10:    $\beta_{N'}(L) \leftarrow \alpha_N(L) + 1$
11:    $\alpha_{N'}(L) \leftarrow m - 1$
12: else
13:    $\beta_{N'}(L) \leftarrow \alpha_N(L-1) + 1$
14:    $\alpha_{N'}(L) \leftarrow \beta_N(L) - 1$
15: end if
16: return $N'$
```

B. ATTRIBUTE ADDITION/SUBTRACTION ALGORITHM

The attribute-based addition/subtraction algorithm relies on comparing the locations of the α-β attribute pairs of both numbers and the current input carry $C_{\text{in}}(i)$. The comparison generates two vectors, X-vector and Y-vector. X-vector for comparing the location of current attributes and Y-vector for comparing the location of the current attributes with the input carry C_{in}. The addition result will be generated based on these two vectors. Table 1 shows the possible values for vector X and Table 2 shows the possible values for vector Y.

The addition/subtraction algorithm proceeds by processing the LSA of both numbers first. The operation will continue processing the attributes until the end of the attributes in one of the inputs. Then, one extra operation is required to deal with the last output carry $C_{\text{out}}(i)$. If the last output carry $C_{\text{out}}(i) = \texttt{NULL}$, the remaining attributes in the none empty number will be appended as MSA to the final result and hence the final C_{out} will be $\texttt{NULL}$. On the other hand, if the last output carry $C_{\text{out}}(i) \neq \texttt{NULL}$, it will be considered as an input carry to the next attribute. The result of this operation will be appended along with the current C_{out} (if not equal $\texttt{NULL}$) as MSA with the remaining attributes in the none empty number.

TABLE 1. *X*−Cases for addition/subtraction function.

Cases	x_1	x_2	x_3	x_4
$\alpha_1(i) > \alpha_2(i)$	1			
$\alpha_1(i) = \alpha_2(i)$	2			
$\alpha_1(i) < \alpha_2(i)$	3			
$\beta_1(i) > \beta_2(i)$		1		
$\beta_1(i) = \beta_2(i)$		2		
$\beta_1(i) < \beta_2(i)$		3		
$\alpha_1(i) > \beta_2(i)$			1	
$\alpha_1(i) = \beta_2(i)$			2	
$\alpha_1(i) < \beta_2(i)$			3	
$\beta_1(i) > \alpha_2(i)$				1
$\beta_1(i) = \alpha_2(i)$				2
$\beta_1(i) < \alpha_2(i)$				3

It should be mentioned that no more operations are required in the case of the current $C_{\text{out}} \neq \texttt{NULL}$ since there should be at least one bit gap between any contiguous attributes in a number. The final C_{out} for the operation will be $\texttt{NULL}$.

Figure 3 shows an example for adding two numbers $N_1 = 2,066,400$ and $N_2 = 262,016$ by using the attribute addition algorithm. The attributes representation for N_1 and N_2 are as follow

$$N_1 \equiv \{(20,15),(10,5)\}$$
$$N_2 \equiv \{(17,7)\}$$

adding these two number requires two iterations. In the first iteration when $i = 1$, the attributes $n_1(i) = (10,5)$ and $n_2(i) = (17,7)$ are added. The initial input carry in this case is $C_{\text{in}} = \texttt{NULL}$. As mentioned earlier, the attribute addition starts by comparing attributes positions and generates two vectors X-vector and Y-vector. In this iteration, since the input carry $C_{\text{in}} = \texttt{NULL}$, only X-vector will be generated. According to Table 1, the X-vector value is $X = [3313]$. The addition result, using Algorithm 4, is

$$N_3'(1) \equiv \{(10,8),(6,5)\}$$
$$C_{\text{out}}'(1) \equiv \{(18,18)\}$$

$N_3'(1)$ is part of the final result of the addition and $C_{\text{out}}'(1)$ will be used as input carry for the next iteration, i.e., its value will be assigned to $C_{\text{in}}''(2)$.

In the second iteration when $i = 2$, there is no more attributes in N_2, so that only Y-vector will be generated. According to Table 2, the Y-vector value is $Y = [11113311]$. The addition result, using Algorithm 4, is

$$N_3''(2) \equiv \{(17,15)\}$$
$$C_{\text{out}}''(2) \equiv \{(21,21)\}$$

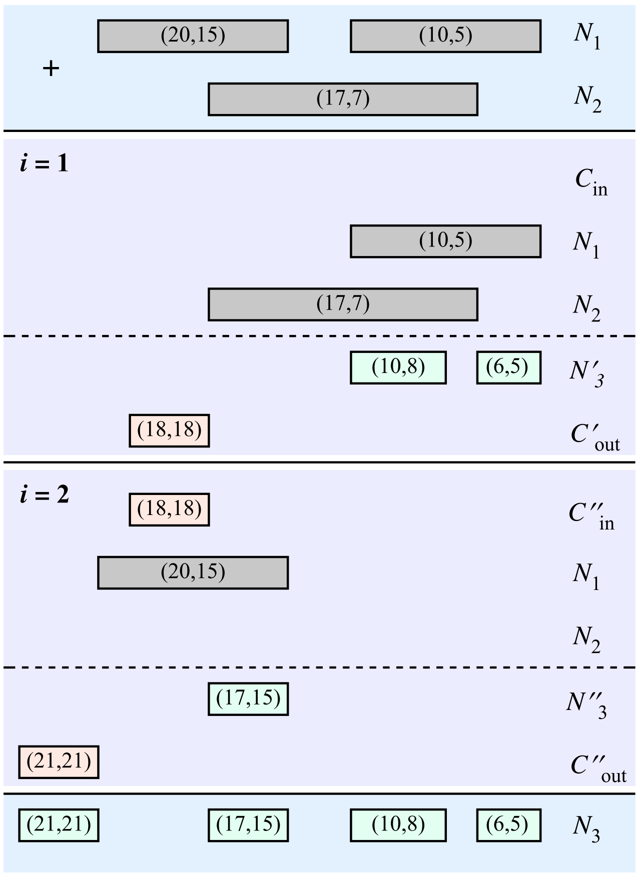


FIGURE 3. Attribute addition example.

$N_3''(2)$ is part of the final result of the addition and its value will be appended as MSA to $N_3'(1)$. Since there are no more attributes in both numbers, $C_{\text{out}}''(2)$ will be appended too to the final result as MSA. The final result for the addition N_3 is as follows

$$N_3 \equiv \{(21,21),(17,15),(10,8),(6,5)\}$$
$$= 2,328,416$$

The pseudo code for adding or subtracting two numbers based on their attributes is shown in Algorithm 4. It is important to point out here that the variables C_{in} and C_{out} are not simple bits but represent attributes with equal values of α and β for each of them.

Lines 2 – 8 setup the input carry C_{in} to the algorithm based on the desired operation add ($s = 0$, $C_{\text{in}} = \texttt{NULL}$) or subtract ($s = 1$, $C_{\text{in}} = (0,0)$). When $s = 1$, the algorithm finds the one's complement using Algorithm 3 and replaces N_2 with the two's complement of N_2.

Lines 9 – 12 deal with the case when either of the numbers N_1 and N_2 are equal to zero. The none zero number will be assigned to N_3 and the output carry C_{out} will be set to $\texttt{NULL}$.

Lines 14 – 19 deal with the case when both numbers are greater than zero. In this case, the algorithm will iterate through attributes in both numbers, two attributes each time,

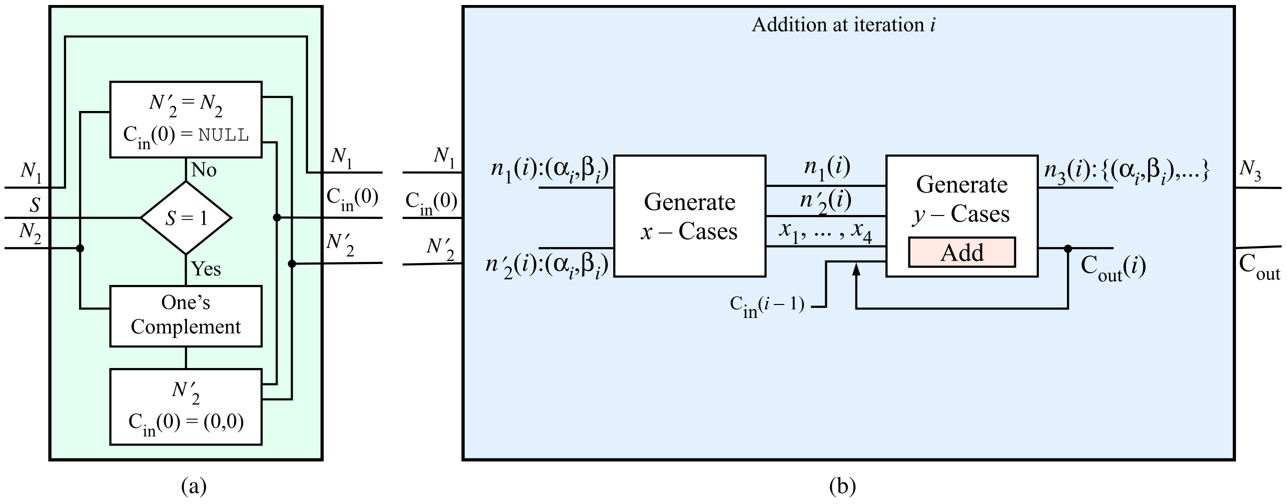


FIGURE 4. Attribute addition and subtraction block diagram. (a) Pre-processing step to select add or add operation. (b) Block diagram for the addition operation at iteration *i*.

TABLE 2. *Y*−Cases for addition/subtraction function.

Cases	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
$\alpha_{in}(i) > \beta_1(i) - 1$	1							
$\alpha_{in}(i) = \beta_1(i) - 1$	2							
$\alpha_{in}(i) < \beta_1(i) - 1$	3							
$\alpha_{in}(i) > \beta_1(i)$		1						
$\alpha_{in}(i) = \beta_1(i)$		2						
$\alpha_{in}(i) < \beta_1(i)$		3						
$\alpha_{in}(i) > \beta_2(i) - 1$			1					
$\alpha_{in}(i) = \beta_2(i) - 1$			2					
$\alpha_{in}(i) < \beta_2(i) - 1$			3					
$\alpha_{in}(i) > \beta_2(i)$				1				
$\alpha_{in}(i) = \beta_2(i)$				2				
$\alpha_{in}(i) < \beta_2(i)$				3				
$\alpha_{in}(i) > \alpha_1(i)$					1			
$\alpha_{in}(i) = \alpha_1(i)$					2			
$\alpha_{in}(i) < \alpha_1(i)$					3			
$\alpha_{in}(i) > \alpha_1(i) - 1$						1		
$\alpha_{in}(i) = \alpha_1(i) - 1$						2		
$\alpha_{in}(i) < \alpha_1(i) - 1$						3		
$\alpha_{in}(i) > \alpha_2(i)$							1	
$\alpha_{in}(i) = \alpha_2(i)$							2	
$\alpha_{in}(i) < \alpha_2(i)$							3	
$\alpha_{in}(i) > \alpha_2(i) - 1$								1
$\alpha_{in}(i) = \alpha_2(i) - 1$								2
$\alpha_{in}(i) < \alpha_2(i) - 1$								3

and add them after generating X-vector and Y-vector by using Generate_X_Cases and Generate_Y_Cases functions respectively. The result of addition $N_3(i)$ is partial result and it may contains more than one attribute. The attributes in $N_3(i)$ will be appended to the final addition result. The output carry $C_{out}(i)$ will be considered as input carry for the next iteration and its value will be assigned to $C_{in}(i+1)$. The algorithm will iterate until it reaches the end of the attributes list in one of the numbers.

If both numbers have the same number of attributes (Lines 20 – 23), the addition operation ends upon reaching the last attribute. If the output carry $C_{out}(i) \neq$ NULL, its value will be appended to N_3 as MSA.

When the final attribute of one of the numbers is reached (Lines 24 – 52), the last step depends on the state of $C_{out}(i)$. When $C_{out}(i)$ = NULL, the remaining attributes in one of the inputs will be appended to N_3. When $C_{out}(i) \neq$ NULL, one extra addition operation is required.

Figure 4 shows an overview of the attribute addition/subtraction operation. Figure 4.a shows the pre-processing step to select add or subtract operation based on the control signal s. This figure corresponds to lines 2 – 8 in Algorithm 4. Figure 4.b is a block diagram for the addition operation at iteration i. The addition operation is shown by the ADD block or the ADD( ) function in Algorithm 4. The add operation depends on the values of the X and Y–Cases summarized in Table 1 and Table 2, respectively. Figure 3 provided a concrete example of the operation of the algorithm.

C. ATTRIBUTE COMPARISON ALGORITHM

In general, comparing two numbers requires determination of their sign and magnitude. Unlike RNS, attribute-based representation allows us to determine the sign and magnitude of a number without conversion to the binary representation. The sign of the number can be determined according to Lemma 3. To compare the magnitudes of the two numbers, we need to compare the MSA of both numbers.

Algorithm 4 Pseudo Code for Attribute Addition and Subtraction (AddSub) for Signed Large Numbers Based on Their Attributes

```
Input: $N_1, N_2, L_1, L_2, s, m$
Output: $N_3, L_3$
 1: $temp \leftarrow 0$; $L_3 \leftarrow 0$;                ▷ Initialization step
 2: if $s = 0$ then
 3:     $C_{in}(0) \leftarrow$ NULL
 4: else $\{s = 1\}$
 5:     $C_{in}(0) \leftarrow (0, 0)$
 6:     $N_2' \leftarrow$ OnesComp($N_2, m$);
 7:     $N_2 \leftarrow N_2'$
 8: end if
 9: if $L_2 = 0$ then
10:     $N_3 \leftarrow N_1$; $C_{out} \leftarrow$ NULL; $L_3 \leftarrow L_1$
11: else if $L_1 = 0$ then
12:     $N_3 \leftarrow N_2$; $C_{out} \leftarrow$ NULL; $L_3 \leftarrow L_2$
13: else $\{L_1 \neq 0$ and $L_2 \neq 0\}$
14:     for $i = 1$ to $\min(L_1, L_2)$ do
15:         $X[1:4] \leftarrow$ Generate_X_Cases($N_1(i), N_2(i)$)
16:         $Y[1:8] \leftarrow$
              Generate_Y_Cases($C_{in}(i-1), N_1(i), N_2(i)$)
17:         $(C_{out}(i), N_3(i), temp) \leftarrow$
                  Add($C_{in}(i-1), N_1(i), N_2(i), X, Y$)
18:         $L_3 \leftarrow L_3 + temp$; $C_{in}(i+1) \leftarrow C_{out}(i)$
19:     end for
20:     if $C_{out}(i) \neq$ NULL and $L_1 = L_2$ then
21:         $N_3 \leftarrow C_{out}(i) \parallel N_3$
22:         $L_3 \leftarrow L_3 + 1$
23:     end if
24:     if $C_{out}(i) \neq$ NULL and $L_1 \neq L_2$ then
25:         $C_{in} \leftarrow$ NULL; $j \leftarrow \min(L_1, L_2) + 1$
26:         if $L_1 > L_2$ then
27:             $X[1:4] \leftarrow$
                    Generate_X_Cases($N_1(j), C_{out}(i)$)
28:             $Y[1:8] \leftarrow$
                 Generate_Y_Cases($C_{in}, N_1(j), C_{out}(i)$)
29:             $(C_{out}(j), N_3(j), temp) \leftarrow$
                      Add($C_{in}, N_1(j), C_{out}(i), X, Y$)
30:             $L_3 \leftarrow L_3 + temp$;
31:             if $C_{out}(j) \neq$ NULL then
32:                 $N_3 \leftarrow N_1(j+1:\text{end}) \parallel C_{out}(j) \parallel N_3$
33:                 $L_3 \leftarrow L_3 + L_1 - j$
34:             else
35:                 $N_3 \leftarrow N_1(j+1:\text{end}) \parallel N_3$
36:                 $L3 \leftarrow L_3 + L_1 - j - 1$
37:             end if
38:         else if $L_1 < L_2$ then
39:             $X[1:4] \leftarrow$
                    Generate_X_Cases($N_2(j), C_{out}(i)$)
40:             $Y[1:8] \leftarrow$
                 Generate_Y_Cases($C_{in}, N_2(j), C_{out}(i)$)
41:             $(C_{out}(j), N_3(j), temp) \leftarrow$
                      Add($C_{in}, N_2(j), C_{out}(i), X, Y$)
42:             $L_3 \leftarrow L_3 + temp$;
```

Continued ►►►

Algorithm 4 (*Continued.*) Pseudo Code for Attribute Addition and Subtraction (AddSub) for Signed Large Numbers Based on Their Attributes

```
43:             if $C_{out}(j) \neq$ NULL then
44:                 $N_3 \leftarrow N_2(j+1:\text{end}) \parallel C_{out}(j) \parallel N_3$
45:                 $L3 \leftarrow L_3 + L_2 - j$
46:             else
47:                 $N_3 \leftarrow N_2(j+1:\text{end}) \parallel N_3$
48:                 $L3 \leftarrow L_3 + L_2 - j - 1$
49:             end if
50:         end if
51:     end if
52: end if
53: return $N_3, L_3$
```

Equality E of the two numbers N_1 and N_2 is determined by the equation:

$$E = \begin{cases} 1 & \text{when} \begin{cases} L_1 = L_2 \\ \alpha_{N_1}(i) = \alpha_{N_2}(i), & \forall\, 1 \leq i \leq L_1 \\ \beta_{N_1}(i) = \beta_{N_2}(i), & \forall\, 1 \leq i \leq L_1 \end{cases} \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

When N_1 and N_2 have opposite signs we have:

$$N_1 > N_2 \text{ when } \alpha_{L_1} < m-1 \text{ and } \alpha_{L_2} = m-1 \tag{24}$$

$$N_1 < N_2 \text{ when } \alpha_{L_1} = m-1 \text{ and } \alpha_{L_2} < m-1 \tag{25}$$

When both N_1 and N_2 are positive, Algorithm 5 is used to determine which number is greater than the other.

When both N_1 and N_2 are negative, Algorithm 5 can still be used to determine which number is greater than the other provided that the L and G outputs are exchanged.

D. ATTRIBUTE MODULAR ADDITION/SUBTRACTION ALGORITHM

When N_1 and N_2 are integers in $GF(p)$, addition and subtraction have to be done modulo p. Modular addition $N_3 = N_1 + N_2 \mod p$ and subtraction $N_3 = N_1 - N_2 \mod p$ can be computed as shown in Algorithm 4 with an additional step for reduction modulo p. The pseudo code to reduce a number modulo p shown in Algorithm 6.

Line 1 assign one to the variable s to put Algorithm 4 in subtraction mode to subtract p when needed. Line 3 compare N against p using algorithm explained in Section IV-C. If the algorithm returns $G = 1$ or $E = 1$, reduction modulo p is needed. The reduction will be done by using Algorithm 4 (Line 5) and the reduced number will be assigned to N'. Otherwise, when the comparison result $L = 1$, no reduction will be needed in this case and the algorithm will assign N to N' (Line 7). The reduction algorithm will iterate until the input number is reduced to a value less than p.

Algorithm 5 Pseudo Code for Comparing two Positive Numbers (COMPARE) Based on Their Attributes

```
Input: $N_1, N_2, L_1, L_2, m$
Output: $G, E, L$
1: $G \leftarrow 0; E \leftarrow 0; L \leftarrow 0$    ▷ Initialization step
2: if $L_1 \neq L_2$ or $E = 0$ then
3:   $i \leftarrow L_1; j \leftarrow L_2$    ▷ MSA first
4:   while $i > 0$ and $j > 0$ do
5:     if $\alpha_{N_1}(i) > \alpha_{N_2}(j)$ then
6:       return $G \leftarrow 1; E \leftarrow 0; L \leftarrow 0$
7:     end if
8:     if $\alpha_{N_1}(i) < \alpha_{N_2}(j)$ then
9:       return $G \leftarrow 0; E \leftarrow 0; L \leftarrow 1$
10:    end if
11:    if $\alpha_{N_1}(i) = \alpha_{N_2}(j)$ then
12:      if $\beta_{N_1}(i) > \beta_{N_2}(j)$ then
13:        return $G \leftarrow 0; E \leftarrow 0; L \leftarrow 1$
14:      end if
15:      if $\beta_{N_1}(i) < \beta_{N_2}(j)$ then
16:        return $G \leftarrow 1; E \leftarrow 0; L \leftarrow 0$
17:      end if
18:      if $\beta_{N_1}(i) = \beta_{N_2}(j)$ then
19:        $i \leftarrow i - 1, j \leftarrow j - 1$
20:        if $\alpha_{N_1}(i) > \alpha_{N_2}(j)$ then
21:          return $G \leftarrow 1; E \leftarrow 0; L \leftarrow 0$
22:        end if
23:        if $\alpha_{N_1}(i) < \alpha_{N_2}(j)$ then
24:          return $G \leftarrow 0; E \leftarrow 0; L \leftarrow 1$
25:        end if
26:        if $\alpha_{N_1}(i) = \alpha_{N_2}(j)$ then
27:          if $\beta_{N_1}(i) > \beta_{N_2}(j)$ then
28:            return $G \leftarrow 0; E \leftarrow 0; L \leftarrow 1$
29:          end if
30:          if $\beta_{N_1}(i) < \beta_{N_2}(j)$ then
31:            return $G \leftarrow 1; E \leftarrow 0; L \leftarrow 0$
32:          end if
33:        end if
34:      end if
35:    end if
36:  end while
37: end if
```

V. LARGE INTEGER ADDITION IN BINARY REPRESENTATION

In this paper we will consider two types of binary adders, the RCA as the baseline and one of the faster parallel prefix adders discussed in Section V-B.

A. USING RIPPLE CARRY ADDITION TECHNIQUE

Regardless of software or hardware implementations, large integers are stored in memory in the form of words. Addition or subtraction operations naturally operate on the words in a sequential fashion due to the carry propagation problem. To ensure fast operations and prevent stalls, blocks of words must be accessed and placed in processor's cache.

Algorithm 6 Pseudo Code to Reduce a Number Modulo p (REDUCE): $N \mod p$

```
Input: $N, p, L_N, L_p, m$
Output: $N'$
1: $s \leftarrow 1$    ▷ Subtraction mode
2: while $N \geq p$ do
3:   $[G, E, L] =$ COMPARE($N$,$p$,$L_N$,$L_p$,$m$)
4:   if $G = 1$ or $E = 1$ then
5:     $N' \leftarrow$ ADDSUB($N$,$p$,$L_N$,$L_p$,$s$,$m$)
6:   else
7:     $N' \leftarrow N$
8:     break;
9:   end if
10: end while
11: return $N'$
```

Algorithm 7 shows the pseudo code for binary addition of m-bits large integers with machine word size is assumed to be w.

Algorithm 7 Pseudo Code for m-Bit Binary Addition (BIADD) for Large Integers

```
Input: $N_1, N_2, w, m,$
Output: $N_3$
1: $C_{\text{in}} \leftarrow 0; C_{\text{out}} \leftarrow 0\ N_3 \leftarrow 0$    ▷ Initialization step
2: $iterations = \lceil m/w \rceil$
3: for $i = 1$ to $iterations$ do
4:   $temp_1 \leftarrow N_1(1 : w)$
5:   $temp_2 \leftarrow N_2(1 : w)$
6:   $(C_{\text{out}}, temp_3) \leftarrow$ ADDWORDS($C_{\text{in}}, temp_1, temp_2$)
7:   $N_3 \leftarrow temp_3 \parallel N_3$
8:   $C_{\text{in}} \leftarrow C_{\text{out}}$
9:   $temp_1 \leftarrow N_1(w + 1 : \text{end})$
10:  $temp_2 \leftarrow N_2(w + 1 : \text{end})$
11: end for
12: if $C_{\text{out}} = 1$ then
13:   $N_3 \leftarrow C_{\text{out}} \parallel N_3$
14: end if
15: return $N_3$
```

Line 2 determines the number of iterations which depends on m and w.

Line 6 is the binary addition method or function AddWords which takes two inputs from the input numbers N_1 and N_2. Addition is done ultimately in hardware using the built-in adder in the machine ALU.

B. USING KOGGE-STONE ADDITION TECHNIQUE

Speed of the add operation is the determining factor that controls the speed of operation of the ALU and the processors. The literature is full of techniques to speed up the classic ripple carry adder. The most promising adders are the binary prefix adders that generate the sum and carry out bits in $\mathcal{O}(\log_2\ m)$ delay. Famous among these adders are

Sklansky adder (SA) [51], Kogge-Stone adder (KSA) [52], Ladner-Fischer adder (LFA) [53] Brent-Kung adder (BKA) [55], and Huan-Carlsson adder (HCA) [56].

Among those adders, Kogge-Stone adder is considered optimal in the sense of its high speed due to its low gate fanout and regular interconnect among the modules. The algorithm for adding two m-bit integers using Kogge-Stone adder is similar to Algorithm 7 except that the function AddWords in Line 6 is now replaced with a w-bit KSA add function whose pseudo code is shown in Algorithm 8.

KSA performs the addition operation in three stages. The first stage is a pre-processing stage. This stage involves parallel computation of propagate P and generate G signals for each pair of bits in augend N_1 and addend N_2 as indicated by Lines 2 – 7 in Algorithm 8. In the second stage, also known as prefix tree, carry propagation is computed in $\log_2 w$ iteration where w is the word size. Each iteration uses P and G from previous iteration to compute the currents ones. The computation in each iteration is parallel and only $w - k$ computations are needed where $1 \leq k \leq 2^{j-1}$ and $2 \leq j \leq \log_2 w + 1$. In the case where P and G are not computed, the value from the previous iteration will be passed to the current iteration. Lines 8 – 19 in Algorithm 8 show the computation in this stage. The last stage is post-processing stage. This stage involves parallel computation of the sum N_3 and the output carry C_{out} as indicated by Lines 20 – 29 in Algorithm 8.

Algorithm 8 Pseudo Code for w-Bit Binary Kogge-Stone Addition (BiAddKsa) for Large Integers

```
Input: $N_1, N_2, C_{\text{in}}, w$
Output: $N_3, C_{\text{out}}$
 1: $C_{\text{out}} \leftarrow 0; N_3 \leftarrow 0$                    ▷ Initialization step
 2: {                                        ▷ Start of parallel code section
 3: for $i = 1$ to $w$ do
 4:     $P(1, i+1) \leftarrow N_1(i) xor N_2(i)$
 5:     $G(1, i+1) \leftarrow N_1(i) and N_2(i)$
 6: end for
 7: }                                        ▷ End of parallel code section
 8: $k \leftarrow 1$
 9: for $j = 2$ to $\log_2 w + 1$ do
10:     {                                    ▷ Start of parallel code section
11:     for $i = 2 + k$ to $w + 1$ do
12:         $P(j, i) \leftarrow P(j-1, i) and P(j-1, i-k)$
13:         $G(j, i) \leftarrow P(j-1, i) and G(j-1, i-k)$ or $G(j-1, i)$
14:     end for
15:     }                                    ▷ End of parallel code section
16:     $k \leftarrow 2^{j-1}$
17: end for
18: $C_{\text{out}}(1) \leftarrow C_{\text{in}}$
19: {                                        ▷ Start of parallel code section
20: for $i = 2$ to $w + 1$ do
21:     $C_{\text{out}}(i) \leftarrow C_{\text{in}} and P(w+1, i)$ or $G(w+1, i)$
22:     $N_3(i-1) \leftarrow C_{\text{out}}(i-1) xor P(1, i)$
23: end for
24: }                                        ▷ End of parallel code section
25: return $N_3, C_{\text{out}}$
```

VI. LARGE INTEGER ADDITION IN RNS REPRESENTATION

The RNS is defined in terms of a pairwise relatively prime moduli set $B = \{b_1, b_2, \ldots, b_L\}$ where B is called base and L is the number of elements in the base (base size). The elements of the set are required to satisfy $GCD(b_i, b_j) = 1$ for $i \neq j$.

An integer X can be represented as $X = \{x_1, x_2, \ldots, x_L\}$ where

$$x_i = X \mod b_i = |X|_{b_i}, \quad 0 \leq x_i < b_i \tag{26}$$

Such a representation is unique for any integer X in the range $0 \leq X < R - 1$, where R is the dynamic range of the moduli set B and is given by

$$R = \prod_{i=1}^{L} b_i \tag{27}$$

Various moduli sets have been proposed for RNS. However, the $3n$-bits dynamic range moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ is widely used RNS moduli set because of its simple and will-formed balanced moduli [65].

Assuming two integers X and Y in RNS representation i.e., $X = \{x_1, x_2, \ldots, x_L\}$ and $Y = \{y_1, y_2, \ldots, y_L\}$. The operation $\circ \in \{+, -, *\}$ can be performed in parallel as follows:

$$X \circ Y = \{|x_1 \circ y_1|_{b_1}, |x_2 \circ y_2|_{b_2}, \ldots, |x_L \circ y_L|_{b_n}\} \tag{28}$$

Equation (28) implies that computations in RNS are performed independently without carry propagation.

Performing arithmetic operations with RNS representation requires conversion from/to binary representations. These conversions introduce an extra overhead on the arithmetic operations. Conversion from binary to RNS representation is a simple process and can be efficiently realized using multi operand modular adders [66].

On the other hand, conversion from RNS to binary representation is a more expensive process and complex to realize [67]. There are several methods proposed for the conversion process from RNS to binary representation such as the Chinese Remainder Theorem (CRT), Mixed-Radix Conversion (MRC) [68], or new Chinese Remainder Theorem (CRT I, CRT II, and CRT III) [69].

The binary equivalent X of a number in RSN representation $\{x_1, x_2, \ldots, x_L\}$ by using CRT method is computed by

$$X = \left(\sum_{i=1}^{L} x_i \, (r_i^{-1} \mod b_i) \, r_i \right) \mod R \tag{29}$$

where $r_i = R/b_i$ and $r_i^{-1} \mod b_i$ is the multiplicative inverse of $r_i \mod b_i$ i.e., $r_i \times r_i^{-1} = 1 \mod b_i$ [4].

Algorithm 9 shows the pseudo code for RNS addition of m-bits large integers.

Lines 2 – 6 to convert N_1 and N_2 from binary representation to their RNS equivalent also called "forward conversion."

Algorithm 9 Pseudo Code for RNS Addition (RNSADD) for Large Integers

```
Input: $N_1, N_2, m, w$
Output: $N_3$
 1: $C_{\text{in}} \leftarrow 0$
 2: $B \leftarrow \{b_1, \ldots, b_L\} = \{2^n + 1, 2^n, 2^n - 1\}$
 3: $n \leftarrow \lceil m/L \rceil$   where $L = \text{size}(B)$
 4: $R \leftarrow \prod_{i=1}^{L} b_i$
 5: $N_1' \leftarrow \{n_1(1), \ldots, n_1(L)\}$
      where $n_1(i) = N_1 \mod b_i, \quad 0 \le n_1(i) < b_i$
 6: $N_2' \leftarrow \{n_2(1), \ldots, n_2(L)\}$
      where $n_2(i) = N_2 \mod b_i, \quad 0 \le n_2(i) < b_i$
 7: {                                   ▷ Start of parallel code section
 8: for $i = 1$ to $L$ do
 9:     $(C_{\text{out}}, N_3') \leftarrow \text{BIADD}(C_{\text{in}}, N_1', N_2', w, n)$
              where $N_3' = \{n_3(1), \ldots, n_3(L)\}$
10: end for
11: }                                   ▷ End of parallel code section
12: $r_i \leftarrow R/b_i$
13: $N_3 \leftarrow \left(\sum_{i=1}^{L} n_3(i)\,(r_i^{-1} \mod b_i)\, r_i\right) \mod R$
14: return $N_3$
```

Lines 7 – 11 to perform the parallel addition operation by using Algorithm 7 where L is the number of elements in the moduli set B and w is the word size to be used in the binary adder.

Lines 12 – 13 to convert N_1 and N_2 from RNS representation to their binary equivalent by using CRT method also called "reverse conversion."

VII. NUMERICAL SIMULATIONS AND SOFTWARE IMPLEMENTATIONS

A. EXPERIMENTAL SETUP

The addition algorithms discussed in Sections IV – VI have been verified through numerical simulations and software implementations using MATLAB version R2018b 64-bit. All simulations were conducted on MacBook Pro with 2.5 GHz Intel Core i7, 16 GB RAM, and running with macOS version 10.14.

Randomly generated integers in the range $[0, 2^m - 1]$ were used as the inputs to the algorithms. The values of m were chosen in the range $2^7 \le m \le 2^{15}$ which correspond to integers with number of bits between 128 to 32,768 bits.

For the simulations, the word size w for the addition algorithms was chosen in the range $4 \le w \le 128$.

B. ESTIMATING THE DELAY OF ADDITION OPERATION

The delays for the addition operation using the three number representations were measured using the timing functions provided in MATLAB.

Attribute-based addition was implemented using Algorithm 4. It should be pointed out here that conversion to and from binary format is not required to implement modular addition/subtraction operations in this representation. This is due to the fact that the comparison operation can be done directly on numbers in the attribute representation according to Algorithm 5.

Binary addition was implemented using Algorithm 7 for RCA binary addition and Algorithm 8 for KSA binary addition. Of course no conversion is necessary here to implement modular addition since the comparison operation is already performed on numbers in binary format.

RNS addition was implemented using Algorithm 9. It should be mentioned that the delay for the RNS adder included the delays associated with the forward and the reverse conversion operations in addition to the parallel addition time. Conversion to and from binary to RNS representations is required in RNS addition to be able to perform the modular addition operation, which requires comparison operation after the RNS add step.

In order to verify the simulation results we perform a complexity analysis of the three addition algorithms to see the effects of m and w on the adder delay.

In this work we normalize all adder delays relative to the delay of the w-bit binary RCA adder. Based on that, the normalized delay of the binary RCA adder is estimated as the number of iterations needed

$$\tau(\text{Binary-RCA}) = \left\lceil \frac{m}{w} \right\rceil \tag{30}$$

The complexity of the normalized binary KSA adder delay is estimated as

$$\tau(\text{Binary-KSA}) = \frac{2 + \log_2 w}{w} \left\lceil \frac{m}{w} \right\rceil \tag{31}$$

The complexity of the normalized RNS adder delay is estimated as

$$\tau(\text{RNS}) = \left\lceil \frac{m}{3w} \right\rceil + T_f + T_r \tag{32}$$

where a three moduli set was assumed and the second term on the RHS represents the normalized conversion delay between RNS and binary representations. T_f is the forward RNS conversion from binary to RNS domain. This delay is essentially a modular reduction step. T_r is the reverse conversion from RNS domain to the binary domain. This delay is the CRT and can be estimated from (29).

The complexity of the normalized attribute adder delay is estimated as

$$\tau(\text{Attribute}) = \left\lceil \frac{\log_2 m}{w} \right\rceil \tag{33}$$

Based on (30), we expect to see a gradual increase in the delay of the binary RCA adder as m increases and the rate of increase gets smaller as w increases.

Based on (31), we expect to see a gradual increase in the delay of the binary KSA adder as m increases and the rate of increase gets smaller as w increases.

Based on (32), we expect similar behavior as the binary adder but the entire delay curve is shifted up by the normalized conversion delays T_f and T_r. Furthermore, these conversion delays might prove a significant factor.

TABLE 3. Speedup of attribute addition relative to RCA binary addition.

m	w (bits)					
(bits)	4	8	16	32	64	128
128	1.21	1.20	0.67	0.43	0.31	0.23
192	1.48	1.90	1.00	0.62	0.40	0.40
224	1.71	1.86	1.08	0.69	0.50	0.34
256	2.09	2.22	1.27	0.79	0.56	0.39
384	2.04	1.59	1.87	1.18	0.80	0.59
512	2.62	2.11	3.18	1.54	1.01	0.75
521	2.68	2.31	2.54	1.51	1.06	0.79
2,048	11.95	9.31	10.40	6.48	4.07	3.00
8,192	69.04	71.86	77.26	43.96	27.13	16.62
32,768	884.32	889.38	909.95	502.13	267.15	144.20

Based on (33), we expect the adder delay to be much smaller than the binary or RNS adders due to the logarithmic term. This effect will become more pronounced for increasing values of m.

C. SIMULATION RESULTS

Comparison between binary, attribute-based and RNS additions was based on comparing the speedup of the attribute-based adder to the other two number formats. Speedup is defined as

$$S_{a-b} = \frac{\tau(\text{binary})}{\tau(\text{attribute})}$$
$$S_{a-r} = \frac{\tau(\text{RNS})}{\tau(\text{attribute})}$$

Table 3 shows the speedup S_{a-b} for the different values of m and w and the binary addition is based on RCA addition. Attribute addition outperforms binary addition for values of $m \geq 2,048$ bits regardless of the values of w. For m in the range $128 \leq m < 2,048$ bits, as m increases attribute addition outperforms binary addition for wider range of w. For all values of $m < 128$, binary addition outperforms attribute addition. Using these values of m in security applications will not provide adequate secure levels.

Internet of things (IoT) devices are characterized by limited computational resources. This translates to machine word sizes between $4 \leq w \leq 16$ bits. Secure IoT devices require high values of m, which demands more from an already limited resource. Table 3 clearly shows the advantage of using attribute-based addition since significantly higher speedup factors translate to shorter computation delays and significantly reduced power consumption. Both these advantages point out the practicality of using attribute-based large integer representations for securing IoT devices.

To meet the challenge of quantum-safe key exchange and encryption, key sizes should exceed 521 bits and more advanced security protocols are used such as SIDH [70]. Table 3 indicates that attribute-based arithmetic has the clear advantage over RCA binary arithmetic. The speedup figures translate directly to reduced computation times and energy consumption.

TABLE 4. Speedup of attribute addition relative to binary-kogge stone addition.

m	w (bits)					
(bits)	4	8	16	32	64	128
128	0.52	0.34	0.11	0.04	0.02	0.01
192	0.50	0.37	0.13	0.05	0.02	0.02
224	0.57	0.43	0.15	0.06	0.04	0.01
256	0.65	0.50	0.17	0.07	0.03	0.02
384	0.64	0.37	0.25	0.10	0.04	0.02
512	0.85	0.46	0.33	0.14	0.06	0.03
521	0.91	0.48	0.33	0.14	0.07	0.03
2,048	3.30	1.90	1.35	0.59	0.35	0.15
8,192	10.38	7.57	5.18	2.04	0.97	0.52
32,768	39.64	26.17	17.56	6.81	3.10	1.52

Table 4 shows the speedup S_{a-b} relative to KSA binary addition for different values of m and w. Attribute addition outperforms binary KSA addition for values of $m \geq 32,786$ bits regardless of the value of w. For m in the range $2,048 \leq m \leq 8,192$ bits, as m increases attribute addition outperforms binary KSA addition for wider range of w. We should point out that the attribute addition algorithm considered here is a sequential algorithm that scans the list of attributes without any attempt at parallelization. Further speedup could be expected when a parallel attribute addition algorithm is developed. Higher values of m is needed to counter the threats of quantum computing attacks. In such situations, our attribute-based addition has the advantage even over KSA binary addition. Finally, in most IoT and personal computing devices, the embedded processors have small value of $w = 8$. In these situations, our attribute-based adders have the clear advantage over all other types of adders.

Table 5 shows the speedup S_{a-r} for different values of m and w when including the delays of converting RNS from/to binary representation. The table clearly shows that attributed addition outperforms RNS addition for all chosen values of m and w. Furthermore, these advantages are retained when meeting the challenges of post quantum computing when key sizes increase.

For completeness, Table 6 shows the speedup S_{a-r} for different values of m and w without including the delays of converting RNS from/to binary representation. The table clearly shows that attributed addition outperforms RNS addition for values of $m \geq 8,192$ (bits) regardless of the values of w. When $m = 2,048$ (bits), attribute addition achieved better performance than RNS addition for values of w within range $4 \leq w \leq 16$. For all values of $m < 2,048$ (bits) RNS addition outperforms attribute addition. This is not a practical advantage however, since RNS addition cannot be used without conversion from/to binary representation for modular addition operation.

Figure 5.a shows adder delay vs. m for the binary RCA, binary KSA, attribute and RNS adders. RNS delay is shown

TABLE 5. Speedup of attribute addition relative to RNS addition with conversions delays.

m	w (bits)					
(bits)	4	8	16	32	64	128
128	516	1,070	1,046	1,055	1,012	1,019
192	525	1,061	1,042	1,076	1,073	1,042
224	491	1,004	977	974	977	988
256	503	1,056	1,006	1,043	1,053	1,064
384	345	526	1,026	1,064	1,022	1,030
512	380	503	1,005	999	1,015	983
521	334	493	986	989	1,020	998
2,048	324	487	1,011	975	1,016	962
8,192	261	515	1,047	1,014	1,018	1,013
32,768	259	478	915	900	884	887

TABLE 6. Speedup of attribute addition relative to RNS addition without conversions delays.

m	w (bits)					
(bits)	4	8	16	32	64	128
128	0.16	0.18	0.11	0.08	0.05	0.06
192	0.26	0.26	0.16	0.11	0.09	0.07
224	0.26	0.30	0.17	0.12	0.09	0.06
256	0.32	0.34	0.23	0.13	0.10	0.07
384	0.30	0.27	0.29	0.19	0.14	0.11
512	0.44	0.33	0.39	0.24	0.17	0.12
521	0.40	0.33	0.37	0.25	0.15	0.12
2,048	1.60	1.28	1.44	0.84	0.55	0.44
8,192	5.82	6.70	7.29	4.92	2.59	1.81
32,768	40.19	41.89	44.24	24.40	14.23	10.02

with and without conversion to and from binary representation. The machine word size in this case was assumed to be $w = 4$ bits.

The red curve with square markers represents the delay of the binary RCA adder. As expected, the binary RCA adder delay increases as the field size m increases due to the increase in the number of iterations required to complete the addition operation according to (30).

The purple curve with right-pointing triangle markers represents the delay for the binary KSA adder. As expected, the binary KSA adder achieved better performance than the binary RCA adder and its delay increases as the field size m increases due to the increase in the number of iterations required to complete the addition operation according to (31).

The green curve with triangle markers shows the delay of the RNS adder with forward and reverse conversions operations. The RNS adder delay shows small dependence on m. Further numerical investigations confirmed that RNS conversion from/to binary representation dominated the actual addition delay. It was also found out that the conversion delay varies very little with the values of m.

The orange curve with diamond markers shows the delay for the RNS adder without forward and reverse conversions operations. As expected, the RNS adder delay increases with m. This is due to the increase in the size of the moduli, which increases the delay of addition in each modulus. However, using RNS adder without conversions operations is not a realistic situation for modular operations and is provided here for the sole purpose of comparison.

The blue curve with circle markers represents the delay of the attribute adder. The delay increases for small values of $m < 512$ bits, and then shows small dependence on m for $m \geq 512$ bits.

The attribute adder achieved better performance than both binary adder and RNS adders with conversions operations delays for all values of m. The attribute adder also achieved better performance than RNS adder without conversions operations for $m \geq 1,230$ bits.

Figure 5.b shows the performance for all adders with word size $w = 8$ bits. The attribute adder outperform binary RCA and RNS adder with conversions operations delays for all values of m. The attribute adder outperform RNS adder without conversions operations delays for values of $m \geq 1,580$ bits. Also, achieved better performance than binary KSA adder for values of $m \geq 512$ bits.

Figure 5.c shows the performance for all adders with word size $w = 16$ bits. The attribute adder outperform binary RCA and RNS adder with conversions operations delays for all values of m. The attribute adder outperform RNS adder without conversions operations delays for values of $m \geq 1,390$ bits. Also, achieved better performance than binary KSA adder for values of $m \geq 750$ bits.

Figure 5.d shows the performance for all adders with word size $w = 32$ bits. The attribute adder outperform RNS adder with conversions operations delays for all values of m. The attribute adder outperform RNS adder without conversions operations delays for values of $m \geq 2,360$ bits. Also, achieved better performance than binary RCA adder for values of $m \geq 326$ bits and better performance than binary KSA adder for values of $m \geq 2,048$ bits.

Figure 5.e shows the performance for all adders with word size $w = 64$ bits. The attribute adder outperform RNS adder with conversions operations delays for all values of m. The attribute adder outperform RNS adder without conversions operations delays for values of $m \geq 3,500$ bits. Also, achieved better performance than binary RCA adder for values of $m \geq 500$ bits and better performance than binary KSA adder for values of $m \geq 3,300$ bits.

Figure 5.f shows the performance for all adders with word size $w = 128$ bits. The attribute adder outperform RNS adder with conversions operations delays for all values of m. The attribute adder outperform RNS adder without conversions operations delays for values of $m \geq 4,600$ bits. Also, achieved better performance than binary RCA adder for values of $m \geq 682$ bits and better performance than binary KSA adder for values of $m \geq 8,192$ bits.

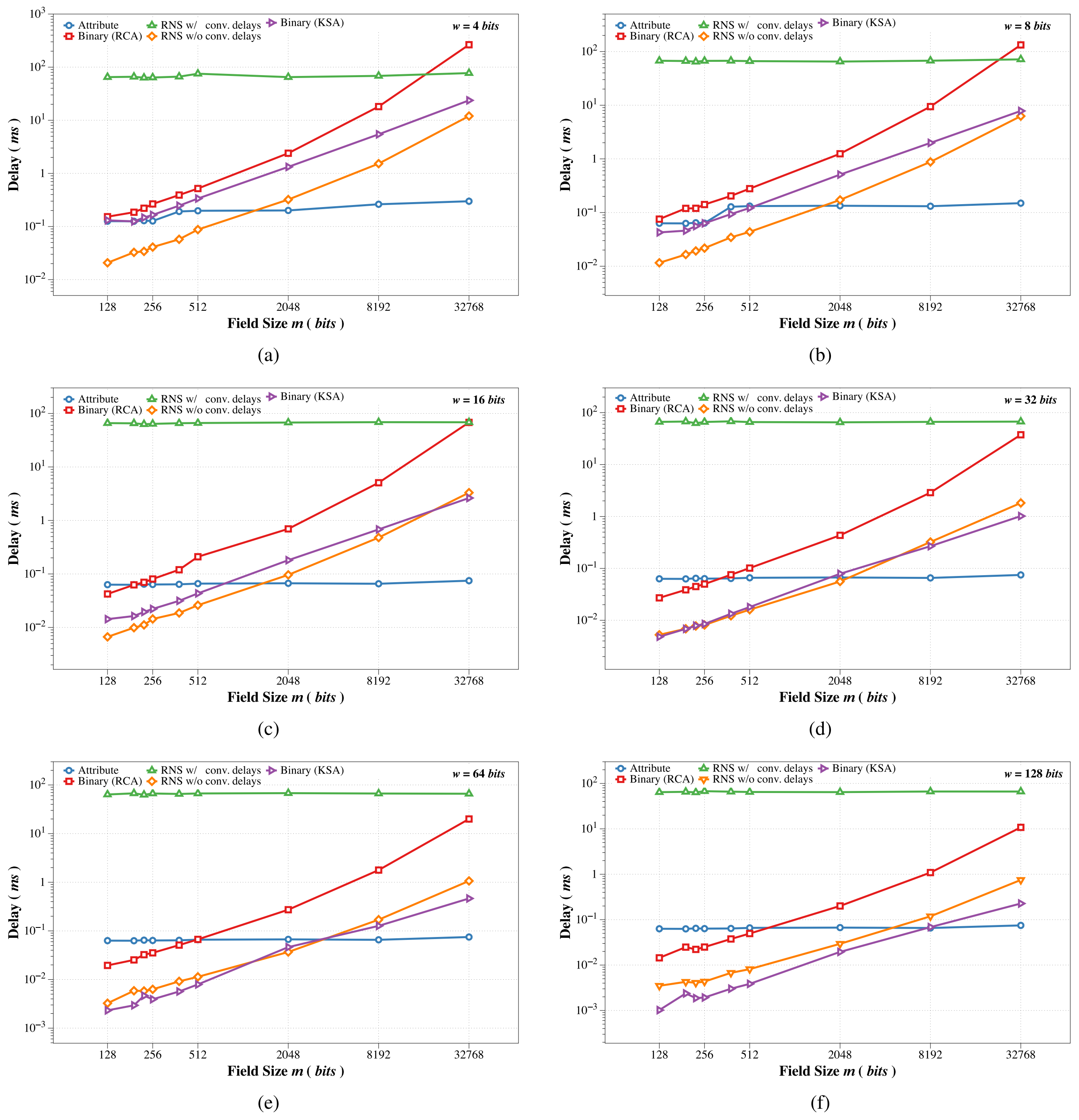


FIGURE 5. Addition delays for m bits field size with: (a) word size $w = 4$ bits, (b) word size $w = 8$ bits, (c) word size $w = 16$ bits, (d) word size $w = 32$ bits, (e) word size $w = 64$ bits, and (f) word size $w = 128$ bits.

VIII. CONCLUSIONS

In this paper, we proposed a new representation for large integers based on their attributes of contiguous ones in their binary representation. We developed algorithms to perform several arithmetic operations based on the new representation such as: two's complement, addition/subtraction, comparison, sign detection, modular addition/subtraction and conversion from binary to attributed-based representation and vice versa. Extensive numerical and software simulations were performed that proved the advantages of our proposed representation in terms of speed over other types of adders including RNS and parallel prefix adders.

There is room for improving our proposed attribute-based addition algorithm in terms of parallelization and developing a hybrid integer representation to include positional and attribute representations. A potential drawback of our proposed number representation is that the attribute representation depends on the number of attributes, which varies widely between one and $m/2$. This leads to highly unpredictable estimation of the adder delay or hardware/software implementations. On the other hand, most cryptographic applications use special primes that happen to have a very small number of attributes between 1 to 5 only. This will produce predictable and very fast modular arithmetic that use our proposed attribute-based representation.

REFERENCES

[1] *Recommended Elliptic Curves for Federal Government Use, Appendix to FIPS 186–4*, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, 2013.

[2] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A faster software implementation of the supersingular isogeny Diffie–Hellman key exchange protocol," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1622–1636, Nov. 2018.

[3] H. L. Garner, "The residue number system," *IRE Trans. Electron. Comput.*, vol. 8, no. 2, pp. 140–147, Jun. 1959.

[4] F. J. Taylor, "Residue arithmetic a tutorial with examples," *Computer*, vol. 17, no. 5, pp. 50–62, May 1984.

[5] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford Univ. Press, 2009.

[6] P. V. A. Mohan, *Residue Number Systems: Theory and Applications*. Cham, Switzerland: Birkhäuser, 2016.

[7] J. Chen and J. Hu, "Energy-efficient digital signal processing via voltage-overscaling-based residue number system," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 7, pp. 1322–1332, Jul. 2013.

[8] C.-H. Chang, A. S. Molahosseini, A. A. E. Zarandi, and T. F. Tay, "Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications," *IEEE Circuits Syst. Mag.*, vol. 15, no. 4, pp. 26–44, 4th Quart., 2015.

[9] T. F. Tay and C.-H. Chang, "A non-iterative multiple residue digit error detection and correction algorithm in RRNS," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 396–408, Feb. 2016.

[10] A. S. Molahosseini, L. S. de Sousa, and C.-H. Chang, Eds., *Embedded Systems Design With Special Arithmetic and Number Systems*. Cham, Switzerland: Springer, 2017.

[11] N. Guillermin, "A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*, in Lecture Notes in Computer Science, vol. 6225, S. Mangard and F.-X. Standaert, Eds. Berlin, Germany: Springer, Aug. 2010, pp. 48–64.

[12] S. Antão and L. Sousa, "The CRNS framework and its application to programmable and reconfigurable cryptography," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 33-1–33-25, Jan. 2013.

[13] P. M. Matutino, R. Chaves, and L. Sousa, "An efficient scalable RNS architecture for large dynamic ranges," *J. Signal Process. Syst.*, vol. 77, nos. 1–2, pp. 191–205, Oct. 2014.

[14] D. Schinianakis and T. Stouraitis, "Multifunction residue architectures for cryptography," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 4, pp. 1156–1169, Apr. 2014.

[15] B. Gérard, J.-G. Kammerer, and N. Merkiche, "Contributions to the design of residue number system architectures," in *Proc. IEEE 22nd Symp. Comput. Arithmetic (ARITH)*, Jun. 2015, pp. 105–112.

[16] J.-C. Bajard, J. Eynard, N. Merkiche, and T. Plantard, "RNS Arithmetic Approach in Lattice-Based Cryptography: Accelerating the 'rounding-off' core procedure," in *Proc. IEEE 22nd Symp. Comput. Arithmetic (ARITH)*, Jun. 2015, pp. 113–120.

[17] K. Bigou and A. Tisserand, "Single base modular multiplication for efficient hardware RNS implementations of ECC," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*, in Lecture Notes in Computer Science, vol. 9293, T. Güneysu and H. Handschuh, Eds. Berlin, Germany: Springer, Sep. 2015, pp. 123–140.

[18] K. Bigou and A. Tisserand, "Hybrid position-residues number system," in *Proc. IEEE 23nd Symp. Comput. Arithmetic (ARITH)*, Jul. 2016, pp. 126–133.

[19] H. T. Vergos, D. Bakalis, and C. Efstathiou, "Fast modulo 2^n+1 multi-operand adders and residue generators," *Integration*, vol. 43, no. 1, pp. 42–48, Jan. 2010.

[20] C. Efstathiou, N. Moschopoulos, K. Tsoumanis, and K. Pekmestzi, "On the design of configurable modulo $2^n\pm1$ residue generators," in *Proc. 15th Euromicro Conf. Digit. Syst. Design (DSD)*, Sep. 2012, pp. 50–56.

[21] A. A. E. Zarandi, A. S. Molahosseini, M. Hosseinzadeh, S. Sorouri, S. Antão, and L. Sousa, "Reverse converter design via parallel-prefix adders: Novel components, methodology, and implementations," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 2, pp. 374–378, Feb. 2015.

[22] A. Hiasat, "A reverse converter and sign detectors for an extended RNS five-moduli set," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 1, pp. 111–121, Jan. 2017.

[23] L. Sousa and P. Martins, "Efficient sign identification engines for integers represented in RNS extended 3-moduli set $\{2^n-1, 2^{n+k}, 2^n+1\}$," *Electron. Lett.*, vol. 50, no. 16, pp. 1138–1139, Jul. 2014.

[24] D. S. Phatak and S. D. Houston, "New distributed algorithms for fast sign detection in residue number systems (RNS)," *J. Parallel Distrib. Comput.*, vol. 97, pp. 78–95, Nov. 2016.

[25] A. Hiasat, "A sign detector for a group of three-moduli sets," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3580–3590, Dec. 2016.

[26] S. Kumar and C.-H. Chang, "A new fast and area-efficient adder-based sign detector for RNS $\{2^n-1, 2^n, 2^n+1\}$," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 7, pp. 2608–2612, Jul. 2016.

[27] S. Bi and W. J. Gross, "The mixed-radix Chinese remainder theorem and its applications to residue comparison," *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1624–1632, Dec. 2008.

[28] Z. Torabi and G. Jaberipur, "Low-power/cost RNS comparison via partitioning the dynamic range," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 5, pp. 1849–1857, May 2016.

[29] H. Xiao, Y. Ye, G. Xiao, and Q. Kang, "Algorithms for comparison in residue number systems," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA)*, Dec. 2016, pp. 1–6.

[30] S. Kumar, C. H. Chang, and T. F. Tay, "New algorithm for signed integer comparison in $\{2^{n+k}, 2^n-1, 2^{n+1}, 2^{n\pm1}-1\}$ and its efficient hardware implementation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 6, pp. 1481–1493, Jun. 2017.

[31] L. Sousa and P. Martins, "Sign detection and number comparison on RNS 3-moduli sets $\{2^n-1, 2^{n+x}, 2^n+1\}$," *Circuits, Syst., Signal Process.*, vol. 36, no. 3, pp. 1224–1246, Mar. 2017.

[32] L. C. Tai and C. F. Chen, "Technical note. Overflow detection in a redundant residue number system," *IEE Proc. E-Comput. Digit. Techn.*, vol. 131, no. 3, pp. 97–98, May 1984.

[33] D. Younes and P. Steffan, "Universal approaches for overflow and sign detection in residue number system based on $2^n-1, 2^n, 2^n+1$," in *Proc. 8th Int. Conf. Syst. (ICONS)*, Jan. 2013, pp. 77–81.

[34] Y. Ye, S. Ma, and J. Hu, "An efficient RNS scaler for moduli set," in *Proc. Int. Symp. Inf. Sci. Eng. (ISISE)*, vol. 2, Dec. 2008, pp. 511–515.

[35] T. F. Tay, C.-H. Chang, and J. Y. S. Low, "Efficient VLSI implementation of 2^n scaling of signed integer in RNS $\{2^n-1, 2^n, 2^n+1\}$," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 10, pp. 1936–1940, Oct. 2013.

[36] L. Sousa, "2^n RNS scalers for extended 4-moduli sets," *IEEE Trans. Comput.*, vol. 64, no. 12, pp. 3322–3334, Dec. 2015.

[37] K. Isupov, V. Knyazkov, and A. Kuvaev, "Fast power-of-two RNS scaling algorithm for large dynamic ranges," in *Proc. 4th Int. Conf. Eng. Telecommun. (EnT)*, Nov. 2017, pp. 135–139.

[38] M. A. Hitz and E. Kaltofen, "Integer division in residue number systems," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 983–989, Aug. 1995.

[39] A. A. Hiasat and H. S. Abdel-Aty-Zohdy, "A high-speed division algorithm for residue number system," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, vol. 3, Apr./May 1995, pp. 1996–1999.

[40] J.-H. Yang, C.-C. Chang, and C.-Y. Chen, "A high-speed division algorithm in residue number system using parity-checking technique," *Int. J. Comput. Math.*, vol. 81, no. 6, pp. 775–780, Jun. 2004.

[41] M. Dasygenis, K. Mitroglou, D. Soudris, and A. Thanailakis, "A full-adder-based methodology for the design of scaling operation in residue number system," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 2, pp. 546–558, Mar. 2008.

[42] Y. Kong and B. Phillips, "Fast scaling in the residue number system," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 3, pp. 443–447, Mar. 2009.

[43] N. I. Cherviakov, M. G. Babenko, and M. N. Shabalina, "Effective implementation of Wang method using approximate method in RNS," in *Proc. IEEE Int. Conf. Soft Comput. Meas. (SCM)*, May 2017, pp. 514–515.

[44] A. Guyot, B. Hochet, and J.-M. Muller, "A way to build efficient carry-skip adders," *IEEE Trans. Comput.*, vol. C-36, no. 10, pp. 1144–1152, Oct. 1987.

[45] N. Burgess, "Accelerated carry-skip adders with low hardware cost," in *Proc. Rec. 35th Asilomar Conf. Signals, Syst. Comput.*, vol. 1, Nov. 2001, pp. 852–856.

[46] B. K. Mohanty and S. K. Patel, "Area–delay–power efficient carry-select adder," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 6, pp. 418–422, Jun. 2014.

[47] U. S. Kumar, K. K. M. Salih, and K. Sajith, "Design and implementation of Carry Select Adder without using multiplexers," in *Proc. IEEE 1st Int. Conf. Emerg. Technol. Trends Electron., Commun. Netw.*, Dec. 2012, pp. 1–5.

[48] S. Jia, S. Lyu, X. Li, L. Liu, and Y. He, "Simplified carry save adder-based array multiplier scheme and circuits design," *Int. J. Circuit Theory Appl.*, vol. 43, no. 9, pp. 1226–1234, May 2015.

[49] R. Zlatanovici, S. Kao, and B. Nikolic, "Energy–delay optimization of 64-bit carry-lookahead adders with a 240 ps 90 nm CMOS design example," *IEEE J. Solid-State Circuits*, vol. 44, no. 2, pp. 569–583, Feb. 2009.

[50] M. Morrison, M. Lewandowski, R. Meana, and N. Ranganathan, "Design of a novel reversible ALU using an enhanced carry look-ahead adder," in *Proc. 11th IEEE Int. Conf. Nanotechnol.*, Aug. 2011, pp. 1436–1440.

[51] J. Sklansky, "Conditional-sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, no. 2, pp. 226–231, Jun. 1960.

[52] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[53] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[54] H. Ling, "High-speed binary adder," *IBM J. Res. Develop.*, vol. 25, no. 3, pp. 156–166, Mar. 1981.

[55] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[56] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proc. IEEE 8th Symp. Comput. Arithmetic (ARITH)*, May 1987, pp. 49–56.

[57] Y. He and C.-H. Chang, "A power-delay efficient hybrid carry-lookahead/carry-select based redundant binary to two's complement converter," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 1, pp. 336–346, Feb. 2008.

[58] A. Ibrahim and F. Gebali, "Optimized structures of hybrid ripple carry and hierarchical carry lookahead adders," *Microelectron. J.*, vol. 46, no. 9, pp. 783–794, Sep. 2015.

[59] X. Cui, W. Liu, S. Wang, E. E. Swartzlander, Jr., and F. Lombardi, "Design of high-speed wide-word hybrid parallel-prefix/carry-select and skip adders," *J. Signal Process. Syst.*, vol. 90, no. 3, pp. 409–419, Mar. 2018.

[60] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1517–1530, Oct. 2014.

[61] N. Poornima and V. S. K. Bhaaskaran, "Area efficient hybrid parallel prefix adders," *Procedia Mater. Sci.*, vol. 10, pp. 371–380, Aug. 2015.

[62] K. Bais and Z. Ali, "Comparison of various adder designs in terms of delay and area," *Int. J. Sci. Res.*, vol. 5, no. 5, pp. 1292–1295, May 2016.

[63] K. C. Shilpa, M. Shwetha, B. C. Geetha, D. M. Lohitha, Navya, and N. V. Pramod, "Performance analysis of parallel prefix adder for datapath VLSI design," in *Proc. 2nd Int. Conf. Inventive Commun. Comput. Technol. (ICICCT)*, Apr. 2018, pp. 1552–1555. [Online]. Available: https://ieeexplore.ieee.org/document/8473087. doi: 10.1109/ICICCT.2018.8473087.

[64] B. K. Joshi, *Data Structure and Algorithm With C*. New York, NY, USA: McGraw-Hill, 2010.

[65] Y. Wang, X. Song, M. Aboulhamid, and H. Shen, "Adder based residue to binary number converters for (2^n-1, 2^n, 2^n+1)," *IEEE Trans. Signal Process.*, vol. 50, no. 7, pp. 1772–1779, Jul. 2002.

[66] K. Navi, A. S. Molahosseini, and M. Esmaeildoust, "How to teach residue number system to computer scientists and engineers," *IEEE Trans. Educ.*, vol. 54, no. 1, pp. 156–163, Feb. 2011.

[67] D. Schinianakis and T. Stouraitis, *RNS-Based Public-Key Cryptography (RSA and ECC)*. Cham, Switzerland: Springer, Mar. 2017, ch. 12, pp. 311–344.

[68] P. V. Ananda Mohan, "RNS to Binary Conversion," in *Residue Number Systems: Theory and Applications*. Cham, Switzerland: Birkhäuser, 2016, ch. 5, pp. 81–132.

[69] Y. Wang, "Residue-to-binary converters based on new Chinese remainder theorems," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 47, no. 3, pp. 197–205, Mar. 2000.

[70] R. Azarderakhsh, D. Jao, and C. Leonardi, "Post-quantum static-static key agreement using multiple protocol instances," in *Proc. Int. Conf. Sel. Areas Cryptogr. (SAC)*, in Lecture Notes in Computer Science, vol. 10719, C. Adams and J. Camenisch, Eds. Cham, Switzerland: Springer, Dec. 2018, pp. 45–63.

BADER ALHAZMI received the B.Sc. degree in electrical and computer engineering from King Abdulaziz University, Jeddah, Saudi Arabia, and the master's degree in information systems security from Concordia University, Montreal, QC, Canada. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Victoria, Canada. His research interests include cryptosystems, hardware security, computer arithmetic, and parallel algorithms.

FAYEZ GEBALI received the B.Sc. degree (Hons.) in electrical engineering from Cairo University, the B.Sc. degree (Hons.) in mathematics from Ain Shams University, and the Ph.D. degree in electrical engineering from the University of British Columbia. He is currently a Professor of electrical and computer engineering with the University of Victoria. His research interests include parallel algorithms, dataflow computing, finite-field arithmetic, and radar signal processing.