

# Fast Low-Power Modulo $2^n + 1$ Squarer Hardware for Efficient Data Processing

Rajashekhar Modugu<sup>1</sup>, Nohpill Park<sup>2</sup>, Yong-Bin Kim<sup>3</sup> and Minsu Choi<sup>1</sup>

<sup>1</sup>Dept of ECE, Missouri University of Science and Technology

Rolla, MO, USA, {rrmt4b, leejongs, choim}@mst.edu

<sup>2</sup>Dept of CS, Oklahoma State University

Stillwater, OK, USA, npark@cs.okstate.edu

<sup>3</sup>Dept of ECE, Northeastern University, Boston, MA, USA, ybk@ece.neu.edu

## Abstract

Modulo  $2^n + 1$  multiplier and squarer are critical components in various applications such as secure communications in networked instrumentation and distributed measurement systems, data encryption and residue arithmetic that increasingly demand high-speed and low-power operations. In this paper, an efficient hardware architecture of modulo  $2^n + 1$  multiplier and squarer are proposed and validated to address the demand. The proposed modulo  $2^n + 1$  implementations have three major functional modules including partial products generation module, partial products reduction module and final stage addition module. The proposed modulo  $2^n + 1$  multiplier and squarer use novel compressor designs and sparse tree adders as primitive building blocks for fast low-power operations. The partial products reduction module is completely redesigned using the novel compressors and the final addition module is implemented using a new sparse tree based inverted end-around-carry adder with significantly less structural complexity. The resulting modulo  $2^n + 1$  multiplier and squarer have been implemented in standard CMOS cell technology and compared both qualitatively and quantitatively with the existing hardware implementations. The unit gate model analysis and the experimental results show that the proposed implementation is faster and consume less power than existing hardware implementations making it a viable option for efficient designs.

## Index Terms

Modulo multiplier, Modulo squarer, Residue Number System (RNS), Compressors, Sparse Tree

Adder.

## I. INTRODUCTION

Modulo arithmetic has been widely used in various applications such as digital signal processing where the residue arithmetic is used for digital filter design [1, 2]. Also, the number of wireless and internet communication nodes has grown rapidly. The confidentiality and the security of the data transmitted over these channels has becoming increasingly important. Cryptographic algorithms such as International Data Encryption Algorithm (IDEA) [5, 6, 7, 8] are frequently used secure communications in the instrumentation and measurement systems. Modulo  $2^n$  addition and modulo  $2^n + 1$  multiplication are the crucial operations in the IDEA algorithm and also modulo  $2^n + 1$  arithmetic operations are used in Fermat number transform computation [4]. Now a days, modulo arithmetic is frequently used in fault tolerant design of ad-hoc networks [3], digital and linear convolution architectures. Apart from these, residue arithmetic is extremely efficient for image processing, speech processing and transforms all of which are extremely important in today's highly dense computing world.

In residue arithmetic, the moduli set  $(2^n - 1, 2^n, 2^n + 1)$  has attracted attention because of it is suitable for effective regular VLSI implementations [9] and easy conversions between binary and residue number system (RNS) [10]. Numerous algorithms and architectures are proposed in the literature for the same moduli set. Using this base, the input operands are  $n - bit$  wide for modulo  $2^n - 1$  and  $2^n$  operations, where as for modulo  $2^n + 1$  operations take inputs with  $n + 1$  bits wide, which makes this modulo operation difficult and calls for special attention. Several architectures and algorithms have been proposed for the design of fast modulo  $2^n + 1$  arithmetic to address this issue [11, 12, 13, 16, 17, 19, 20]. Modulo  $2^n + 1$  has found many applications in Fermat Number Transformation and IDEA [4, 21]. The ability to perform fast modulo  $2^n + 1$  is then still a major challenge, particularly from a hardware point of view. Even though a modulo  $2^n + 1$  multiplier can be implemented using look-up tables, the memory requirements are a big constraint for large values of  $n$ . Hence, to avoid the exponential growth of the memory requirements several implementations based on combinational arithmetic circuits have been proposed. A few of these implementations are described briefly below.

In the work of Wrzyszc and Milford [13], the  $n + 1 \times n + 1$  partial product matrix of the modulo multiplier is transformed into an  $n \times n$  partial product matrix. This transformation is

based on the observation that the most significant bit of the input operands is 1 only when the inputs are 1s. And also, the periodic properties of  $|2^i|_{2^n+1}$  are employed through out the computation process. Although this multiplier has a series of three  $n$ -bit adders and multiplexers in the final stage, it has a considerably regular structure that is suitable for VLSI implementation.

To overcome the problem of  $n + 1$  bit input length, diminished-1 representation has been used and applied in some of the recent implementations. For diminished-1 operand representation, bit-pair Booth recoding technique was used in Ma's work [14] to reduce the number of partial products to approximately  $n/2$  at the cost of additional Carry Save Adders (CSA). In this work, a radix-4 Booth recoding technique and modulo CSA modules are used to reduce the partial products in diminished-1 representation. Zimmerman, in his work [16], proposed a modulo  $2^n + 1$  multiplier with weighted operand representation that can be adopted to diminished-1 operand representation. Wallace tree and Booth encoding are applied to this multiplier to speed up the operation. However, additional circuitry for the  $2^n$  correction is required in this implementation and considered to be significant overhead to this multiplier. In the work of Wang et. al. [19] the diminished-1 number representation is used along with Wallace tree architecture. In this work, modified Booth recoding scheme is used to speed up the reduction of the partial products and the correction of the final stage addition results. This method has zero handling property but it requires extra circuits to calculate correction factors. Based on the observations made by Wrzyszc and Milford [13], Vergos and Efstathiou et. al. [20] proposed another novel modulo multiplier which has only one correction factor. In this multiplier design, the final stage modulo  $2^n + 1$  addition is converted into modulo  $2^n$  addition using a part of the total correction factor of the multiplier. This multiplier with weighted inputs has zero handling property without any extra hardware and does not require any extra encoding either. In the work of Piestrak [37], modulo squarers with base  $(2^n + 1, 2^n - 1)$  are described, in these implementations partial products are reduced using carry save adder and the final output is produced using end around carry propagate adder. The modulo  $2^n + 1$  squarer algorithm is efficiently designed with carry save adders and parallel prefix based end around carry adder. But, the inputs and outputs are in diminished-1 representation, which require conversions from/to weighted number system.

Even though the multiplier and squarers proposed in [20, 36] achieved considerable enhancements over the existing designs in terms of power consumption and delay, hardware implementation of this multiplier requires a special attention. The critical path of this multiplier

depends on the partial products reduction stage which uses a Carry Save Adder (CSA) and on the final stage addition. Hence, efficient design of Carry Save adder and the final stage adder is of high need.

In order to address this issue, a new efficient hardware design for modulo  $2^n + 1$  multiplier has been proposed and validated in this work.

The rest of this manuscript is organized as follows. In section II, novel multiplexor-based compressors are briefly reviewed. Section III presents the algorithm from the work of Vergos and Efstathiou [20] which is used to implement the multiplier and an efficient squarer algorithm is described based on the multiplier algorithm. In section IV, the proposed hardware implementation of the modulo  $2^n + 1$  multiplier/squarer and novel sparse tree based inverted End Around Carry (EAC) adder are described. Qualitative and quantitative comparisons of the proposed multiplier and squarer with the well-known existing implementations are discussed in section V. In section VI conclusions are drawn.

## II. PRELIMINARIES AND REVIEWS

### A. Compressors for high-speed arithmetic circuits

1) *MUX vs XOR*: Multiplexor (MUX) is one of the logic gates used extensively in the digital design, which is very useful in efficient design of arithmetic and logic circuits. According to the CMOS implementation of MUX [22], it performs better in terms of power and delay compared to exclusive-OR (XOR). Suppose,  $X$  and  $Y$  are inputs to the XOR gate, the output is  $X\bar{Y} + \bar{X}Y$ . The same XOR can be implemented using MUX with inputs  $X, \bar{X}$  and select bit  $Y$ . Efficient compressors have been designed using MUX and reported in [23]. In the proposed compressors, both output and its complement of these gates are used simultaneously and therefore result in a reduced number of garbage outputs. Existing CMOS designs of 2:1 MUX and 2-input XOR are shown in Fig. 1 for comparison.

2) *Compressors for Arithmetic Circuits*: A  $(p, 2)$  compressor has  $p$  inputs  $X_1, X_2 \dots X_{p-1}, X_p$  and two output bits (i.e., Sum bit and Carry bit) along with carry input bits and carry output bits. Its functionality can be represented by the following equation:

$$\sum_{i=1}^p X_i + \sum_{i=1}^t (C_{in})_i = Sum + 2(Carry + \sum_{i=1}^p (C_{out})_i)$$

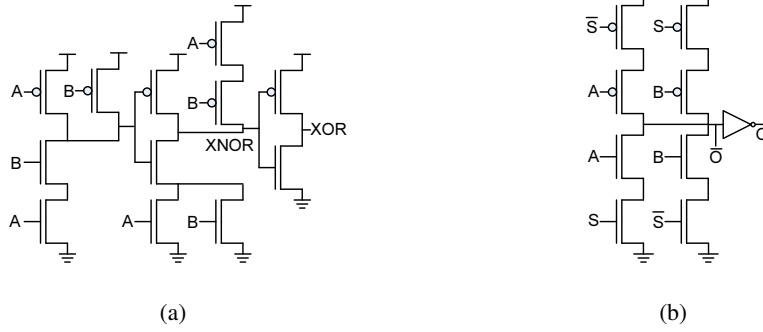


Fig. 1. CMOS implementation of 2-input (a) XOR (b) MUX

For example, a  $(5, 2)$  compressor takes 5 inputs and 2 carry inputs and a  $(7, 2)$  compressor takes 7 inputs and 2 carry inputs. Block diagrams of 5:2 and 7:2 compressors are shown in Fig. 2. Efficient designs of the existing XOR-based 5:2 and 7:2 compressors have critical path delays of  $4\Delta(XOR)$  and  $6\Delta(XOR)$  (delay denoted by  $\Delta$ ), respectively [24, 25].

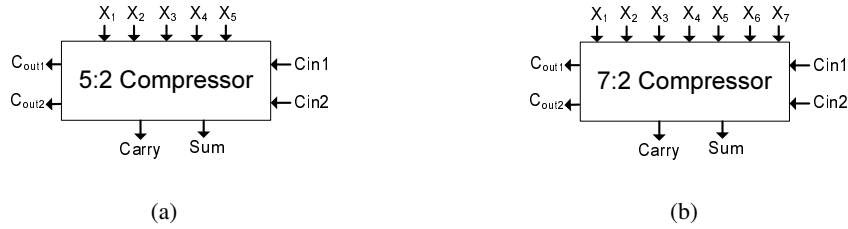


Fig. 2. Block diagram of (a) 5:2 compressor (b) 7:2 compressor

The newly proposed efficient compressors use multiplexers in place of XOR gates, resulting in high speed arithmetic due to reduced gate delays [23]. Also as shown in Fig. 1, in all the existing CMOS implementations of the XOR and MUX gates both the output and its complement are available but the designs of compressors available in literature do not use these outputs efficiently. In CMOS implementation of the MUX if both the select bit and its complement are generated in the previous stage then its output can be generated with much less delay because the switching of the transistor is already completed. And also if both the select bit and its complement are generated in the previous stage then the additional stage of the inverter can be eliminated which reduces the overall delay in the critical path. The proposed MUX-based 5:2 [23] and 7:2 compressors are shown in Fig. 3, the critical path delays of which are  $\Delta(XOR) + 3\Delta(MUX)$  and  $\Delta(XOR) + 5\Delta(MUX)$ , respectively (where  $\Delta(x)$  is the delay of

gate  $x$ ). CGEN block used in the non-critical path shown in this figure can be obtained from the equation  $C_{outi} = (X + Y) \cdot Z + X \cdot Y$ .

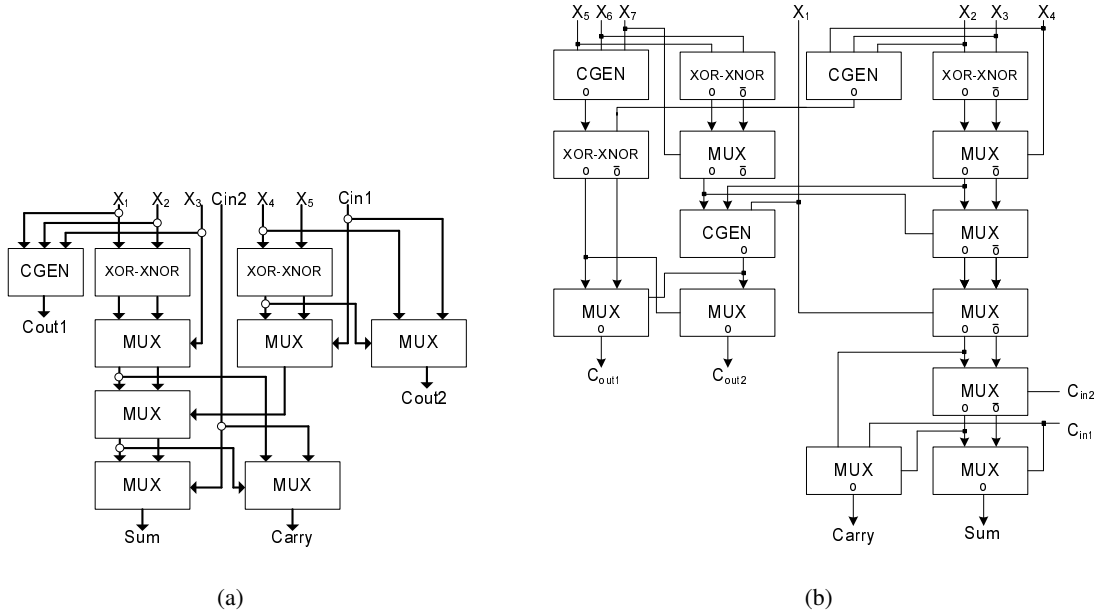


Fig. 3. Proposed MUX-based design of (a) 5:2 compressor [23] (b) 7:2 compressor

### III. ALGORITHM FOR IMPLEMENTATION OF MODULO $(2^n + 1)$ MULTIPLIER

Based on the following description of the algorithm for computation of  $X \cdot Y \bmod (2^n + 1)$ , a new algorithm for modulo  $2^n + 1$  squarer is presented below. An efficient modulo  $2^n + 1$  squarer algorithm is presented in [36] using diminished-1 representation. The disadvantage of the algorithm is the conversions from/to diminished-1 system to/from weighted number system. Hence, based on the multiplier algorithm which uses weighted number system, modulo  $2^n + 1$  squarer for weighted number system is proposed. From the architectural characteristic comparisons [20], the algorithm presented in [20] is considered as the best existing algorithm for the computation of  $X \cdot Y \bmod (2^n + 1)$  in the literature. According to the algorithm it takes two  $n + 1$  bit unsigned numbers as inputs and gives one  $n + 1$  bit unsigned number as output. The proposed implementation can be adapted to IDEA cipher [8, 5, 21], in which the modulo  $(2^n + 1)$  multiplication module takes two  $n$ -bit inputs and gives one  $n$ -bit output, by assigning the most significant bits of the inputs as zeros.

In the following text, modulo  $2^n + 1$  multiplier algorithm is described and modulo  $2^n + 1$  squarer is deduced from this. Let  $|A|_B$  denote the residue of  $A$  modulo  $B$ . Let  $X$  and  $Y$  be two inputs represented as  $X = x_n x_{n-1} \dots x_0$  and  $Y = y_n y_{n-1} \dots y_0$  where the most significant bits  $x_n$  and  $y_n$  are ones only when the inputs are  $2^n$  and  $2^n$ , respectively.  $|X \cdot Y|_{2^n+1}$  can be represented as follows:

$$\begin{aligned} P &= |X \cdot Y|_{2^n+1} = \left| \sum_{i=0}^n x_i 2^i \cdot \sum_{j=0}^n y_j 2^j \right|_{2^n+1} \\ &= \left| \sum_{i=0}^n \left( \sum_{j=0}^n p_{i,j} 2^{i+j} \right) \right|_{2^n+1} \end{aligned} \quad (1)$$

where  $p_{i,j} = x_i \text{ AND } y_j$ .

The  $n \times n$  partial product matrix shown in Fig. 6 is derived from the initial partial product matrix in Fig. 4, based on several observations. The first observation is as follows. The initial partial product matrix can be divided into four groups  $A$ ,  $B$ ,  $C$  and  $D$  in which the terms in only one group can be different from '0'. Groups  $A$ ,  $B$ ,  $D$  and  $C$  are different from '0', if inputs  $(X, Y)$  are in the form of  $(0Z, 0Z)$ ,  $(1Z, 0Z)$ ,  $(0Z, 1Z)$  and  $(10 \dots 0, 10 \dots 0)$ , respectively (i.e., 'Z' is a 16-bit vector here). Hence the four groups can be integrated into a single group by performing logical OR operation (denoted by  $\vee$  in Fig. 5) instead of arithmetically adding the bits from different groups. Logical OR operation is performed on the terms of the groups  $B$ ,  $D$  and  $A$  in the columns with weight  $2^n$  up to  $2^{2n-2}$  and on the two terms of the groups  $B$  and  $D$  with weight  $2^{2n-1}$  ( $q_i$  represents the ORed terms of the groups  $B$  and  $D$  which are in the form of  $p_{n,i}$  and  $p_{i,n}$ , where  $q_i = p_{n,i} \vee p_{i,n}$ ). Since  $|2^{2n-1}|_{2^n+1} = 2^{n-1} + 1$ , the term with weight  $2^{2n-1}$ ,  $q_{n-1}$  can be substituted by two terms  $q_{n-1}$  in the columns with weight  $2^{n-1}$  and 1 respectively, and ORed with any term of the group  $A$  there. Moreover, since  $|2^{2n}|_{2^n+1} = 1$ , the term  $p_{n,n}$  can be ORed with any term in the column with weight  $2^0$ , in our case it is ORed with  $p_{0,0}$ . The modified partial product matrix is shown in Fig. 5.

The second observation is about repositioning of the partial product terms in the modified partial product matrix, with weight greater than  $2^{n-1}$  based on the following equation:

$$\begin{aligned} |s2^i|_{2^n+1} &= |-s2^{|i|_n}|_{2^n+1} = |(2^n + 1 - s)2^{|i|_n}|_{2^n+1} \\ &= |\bar{s}2^{|i|_n} + 2^n 2^{|i|_n}|_{2^n+1} \end{aligned} \quad (2)$$

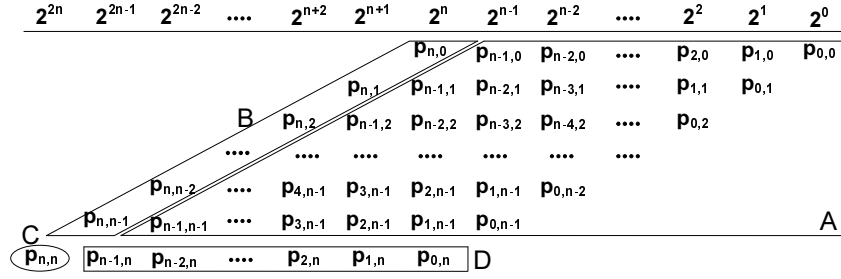


Fig. 4. Initial partial product matrix

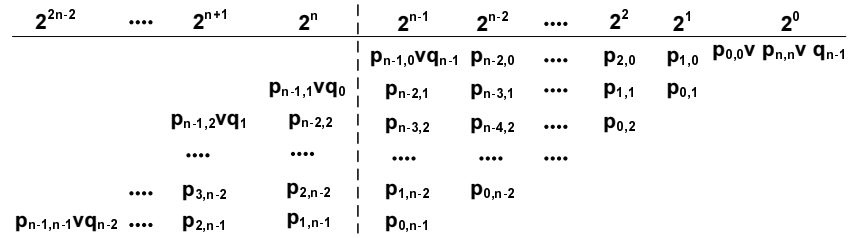


Fig. 5. Modified partial product matrix

Equation (2) shows that the repositioning of each bit to  $i^{th}$  bit position results in a correction factor of  $2^n 2^{|i|n}$ . In the first partial product vector, there is no bit with weight greater than  $2^{n-1}$  to be repositioned, in the second partial product vector one bit need to be repositioned resulting in a correction factor of  $2^0 2^n = (2^1 - 1)2^n$  and in the third partial product vector two bits need to be repositioned with correction factor  $(2^0 + 2^1)2^n = (2^2 - 1)2^n$  and so on. Hence the correction factor for the entire partial product matrix would be:

$$\begin{aligned}
 COR_1 &= 2^n [(2^1 - 1) + (2^2 - 1) + \dots + (2^{n-1} - 1)] \\
 &= 2^n [2(1 + 2 + 2^2 + \dots + 2^{n-2}) - (n - 1)] \\
 &= 2^n (2^n - n - 1)
 \end{aligned} \tag{3}$$

The  $n \times n$  partial product matrix along with the equation (3) results in  $n + 1$  partial product vectors. These  $n + 1$  partial products are to be modulo  $2^n + 1$  added to produce two final vectors (i.e., Sum vector and Carry vector). This can be done using a Carry Save Adder (CSA) with  $(n - 1)$  levels. As the CSA works as a modulo  $2^n + 1$  adder, the carry out at each level of the CSA has to be fed back as the carry-in of the next subsequent level. Suppose the carry-out bit

$2^{n-1}$	$2^{n-2}$	$2^{n-3}$	$2^2$	$2^1$	$2^0$
$PP_0 = p_{n-1,0} \vee q_{n-1}$	$p_{n-2,0}$	$p_{n-3,0}$	$p_{2,0}$	$p_{1,0}$	$p_{0,0} \vee q_{n-1} \vee p_{n,n}$
$PP_1 = p_{n-2,1}$	$p_{n-3,1}$	$p_{n-4,1}$	$p_{1,1}$	$p_{0,1}$	$p_{n-1,1} \vee q_0$
$PP_2 = p_{n-3,1}$	$p_{n-4,2}$	$p_{n-5,2}$	$p_{0,2}$	$p_{n-1,2} \vee q_1$	$p_{n-2,2}$
....	....	....	....	....	....
$PP_{n-2} = p_{1,n-2}$	$p_{0,n-2}$	$p_{n-1,n-2} \vee q_{n-3}$	$p_{4,n-2}$	$p_{3,n-2}$	$p_{2,n-2}$
$PP_{n-1} = p_{0,n-1}$	$p_{n-1,n-1} \vee q_{n-2}$	$p_{n-2,n-1}$	$p_{3,n-1}$	$p_{2,n-1}$	$p_{1,n-1}$

Fig. 6. Final  $n \times n$  partial product matrix

of the  $n^{th}$  column at  $i^{th}$  level of CSA is  $c_i$  with weight  $2^n$ , this carry-out can be reduced to:

$$|c_i 2^n|_{2^{n+1}} = |-c_i|_{2^{n+1}} = |2^n + \bar{c}_i|_{2^{n+1}}$$

Therefore carry output bits from the  $n^{th}$  column at each level can be used as carry input bits of the next level. In an  $n - 1$  stage, CSA produces  $n - 1$  such carry-out bits [20]. Hence there is a second correction factor (4). The final correction factor due to the carry out bits of the CSA is:

$$COR_2 = |2^n(n - 1)|_{2^{n+1}} \quad (4)$$

The final correction factor can be calculated by summing up  $COR_1$  and  $COR_2$  as follows:

$$\begin{aligned} COR &= COR_1 + COR_2 \\ &= |2^n(n - 1) + 2^n(2^n - n - 1)|_{2^{n+1}} \\ &= |2^n(2^n - 2)|_{2^{n+1}} = 3 \end{aligned} \quad (5)$$

Therefore, the constant '3' in equation (5) is the final correction factor.

For modulo  $2^n + 1$  squarer, the inputs in (1) are equal are let it be represented by  $X$  and  $X = x_n x_{n-1} \dots x_0$ . The equation (1) is transformed into the following equation.

$$\begin{aligned} P &= |X \cdot Y|_{2^{n+1}} = \left| \sum_{i=0}^n x_i 2^i \cdot \sum_{j=0}^n x_j 2^j \right|_{2^{n+1}} \\ &= \left| \sum_{i=0}^n \left( \sum_{j=0}^n p_{i,j} 2^{i+j} \right) \right|_{2^{n+1}} \end{aligned}$$

where  $p_{i,j} = x_i \text{ AND } x_j$ .

$2^{n-1}$	$2^{n-2}$	$2^{n-3}$	....	$2^2$	$2^1$	$2^0$
$PP_0 = p_{n-1,0} \vee p_{n,n-1}$	$p_{n-2,0}$	$p_{n-3,0}$	....	$p_{2,0}$	$p_{1,0}$	$p_{0,0} \vee p_{n,n-1} \vee p_{n,n}$
$PP_1 = p_{n-2,1}$	$p_{n-3,1}$	$p_{n-4,1}$	....	$p_{1,1}$	$p_{0,1}$	$p_{n-1,1} \vee p_{n,0}$
$PP_2 = p_{n-3,2}$	$p_{n-4,2}$	$p_{n-5,2}$	....	$p_{0,2}$	$p_{n-1,2} \vee p_{n,1}$	$p_{n-2,2}$
$PP_3 = p_{n-4,3}$	$p_{n-5,3}$	$p_{n-6,3}$	....	$p_{n-1,3} \vee p_{n,2}$	$p_{n-2,3}$	$p_{n-3,3}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$PP_{n-3} = p_{n-3,2}$	$p_{n-3,1}$	$p_{n-3,0}$	....	$p_{n-3,5}$	$p_{n-3,4}$	$p_{n-3,3}$
$PP_{n-2} = p_{n-2,1}$	$p_{n-2,0}$	$p_{n-1,n-2} \vee p_{n,n-3}$	....	$p_{n-2,4}$	$p_{n-2,3}$	$p_{n-2,2}$
$PP_{n-1} = p_{n-1,0}$	$p_{n-1,n-1} \vee p_{n,n-2}$	$p_{n-1,n-2}$	....	$p_{n-1,3}$	$p_{n-1,2}$	$p_{n-1,1}$

Fig. 7.  $n \times n$  partial product matrix of modulo  $2^n + 1$  squarer obtained from Fig.6

In the above equation (III),  $p_{i,j} = x_i \text{ AND } x_j$  and  $p_{j,i} = x_j \text{ AND } x_i$ . Therefore, the two terms  $p_{i,j}$  and  $p_{j,i}$  are equivalent. The final partial products matrix of modulo  $2^n + 1$  multiplier shown in Fig.6, can be applied to modulo squarer by making the following changes.

- Replacing  $p_{j,i}$  with  $p_{i,j}$  since  $x_i \text{ AND } x_j = x_j \text{ AND } x_i$ .
- Since  $q_k = p_{n,k} \vee p_{k,n}$  and  $p_{n,k} \vee p_{k,n} = p_{n,k} \vee p_{n,k} = p_{n,k}$ ,  $q_k$  can be substituted with  $p_{n,k}$ .
- $p_{k,k} = x_k \text{ AND } x_k = x_k$ .

The resulting partial product matrix for the modulo  $2^n + 1$  after making the above changes is shown in Fig.7. From Fig. 7 we can observe that some appear twice in the same column as  $p_{i,j}$  or  $\overline{p_{i,j}}$ . Such pairs can be presented as a single term  $2(p_{i,j})$ , since  $p_{i,j} + p_{i,j} = 2(p_{i,j})$ . The term  $2(p_{i,j})$  in the column with weight  $2^{i+j}$  can be placed in column with weight  $2^{i+j+1}$  as  $p_{i,j}$ . In each column, pair of partial product bits with this property are shifted to the left column. Using the explanation given to compute  $COR_2$ , the paired terms ( $2(p_{i,j})$ ) in the leftmost column are placed in the rightmost column resulting in a correction factor of  $2^n$  per one shifted term.

The number of terms to be shifted from a column to its left column depends on the value of  $n$ . When  $n$  is even then, then columns with weight  $2^{i+j}$  and  $i + j \in 0, 2, 4 \dots n - 2$  have  $\frac{n-4}{2}$  pairs of equal terms, and when  $i + j$  is odd i.e.,  $i + j \in 1, 3, 5 \dots n - 1$  have  $\frac{n-4}{2} + 1$  pairs of equal terms. When  $n$  is odd each column of the partial products matrix has  $\frac{n-3}{2}$  pairs of equal terms. These changes in the partial product matrix need recalculation of the correction factors for modulo  $2^n + 1$  squarer. The modulo  $2^n + 1$  squarer has three correction factors. First correction factor is  $COR_{1\_Squarer}$  and it remains same as it is derived from the repositioning of the partial product bits from the initial partial product matrix. The second correction factor ( $COR_{2\_Squarer}$ ) is computed based on the number of partial products in the final partial products matrix. Since, in the final partial products matrix of the squarer, we have varied number of partial products

depending on  $n$  and including the correction factor the total number of partial products is given by.

$$\text{Number of partial products} = \begin{cases} n - \frac{n-4}{2} + 1 & \text{when } n \text{ is even} \\ n - \frac{n-3}{2} + 1 & \text{when } n \text{ is odd} \end{cases}$$

Including the correction factor we have  $\frac{n+6}{2}$ ,  $\frac{n+5}{2}$  number of partial products when  $n$  is even and odd respectively. These partial products have to be modulo  $2^n + 1$  added to produce the final Sum and Carry vectors. These partial products can be reduced using a CSA network of  $\frac{n+6}{2} - 2 = \frac{n+2}{2}$  full adders in a single column when  $n$  is even and  $\frac{n+5}{2} - 2 = \frac{n+1}{2}$  full adders when  $n$  is odd. As explained earlier, the carry out bits of the full adders in the  $n^{\text{th}}$  column with weight  $2^n$  of the CSA network have to be fed back as the carry inputs to the column with weight  $2^0$ . The correction factor  $COR_{2\_Squarer}$  caused by this operation is given by.

$$COR_{2\_Squarer} = \begin{cases} 2^n \left(\frac{n+2}{2}\right) & \text{when } n \text{ is even} \\ 2^n \left(\frac{n+1}{2}\right) & \text{when } n \text{ is odd} \end{cases}$$

In modulo  $2^n + 1$  implementation another correction factor  $COR_{3\_Squarer}$  is possible as pair of partial product equal terms from the leftmost column of the partial products matrix are shifted to the rightmost column. Each shifted term results in a correction factor of  $2^n$  and the correction factor ( $COR_{3\_Squarer}$ ) is given by.

$$COR_{3\_Squarer} = \begin{cases} 2^n \left(\frac{n-4}{2}\right) & \text{when } n \text{ is even} \\ 2^n \left(\frac{n-3}{2}\right) & \text{when } n \text{ is odd} \end{cases}$$

The total correction factor for a modulo  $2^n + 1$  squarer can be obtained by summing up all the above three correction factors.

$$\begin{aligned} COR_{Squarer} &= COR_{1\_Squarer} + COR_{2\_Squarer} + COR_{3\_Squarer} \\ &= \begin{cases} \left| 2^n(2^n - n - 1) + 2^n \left(\frac{n+2}{2}\right) + 2^n \left(\frac{n-4}{2}\right) \right|_{2^{n+1}} & \text{when } n \text{ is even} \\ \left| 2^n(2^n - n - 1) + 2^n \left(\frac{n+1}{2}\right) + 2^n \left(\frac{n-3}{2}\right) \right|_{2^{n+1}} & \text{when } n \text{ is odd} \end{cases} \quad (6) \\ &= |2^n(2^n - 2)|_{2^{n+1}} = 3 \end{aligned}$$

From (6), we can observe that the total correction factors for modulo  $2^n + 1$  multiplier and modulo  $2^n + 1$  squarer are the same.

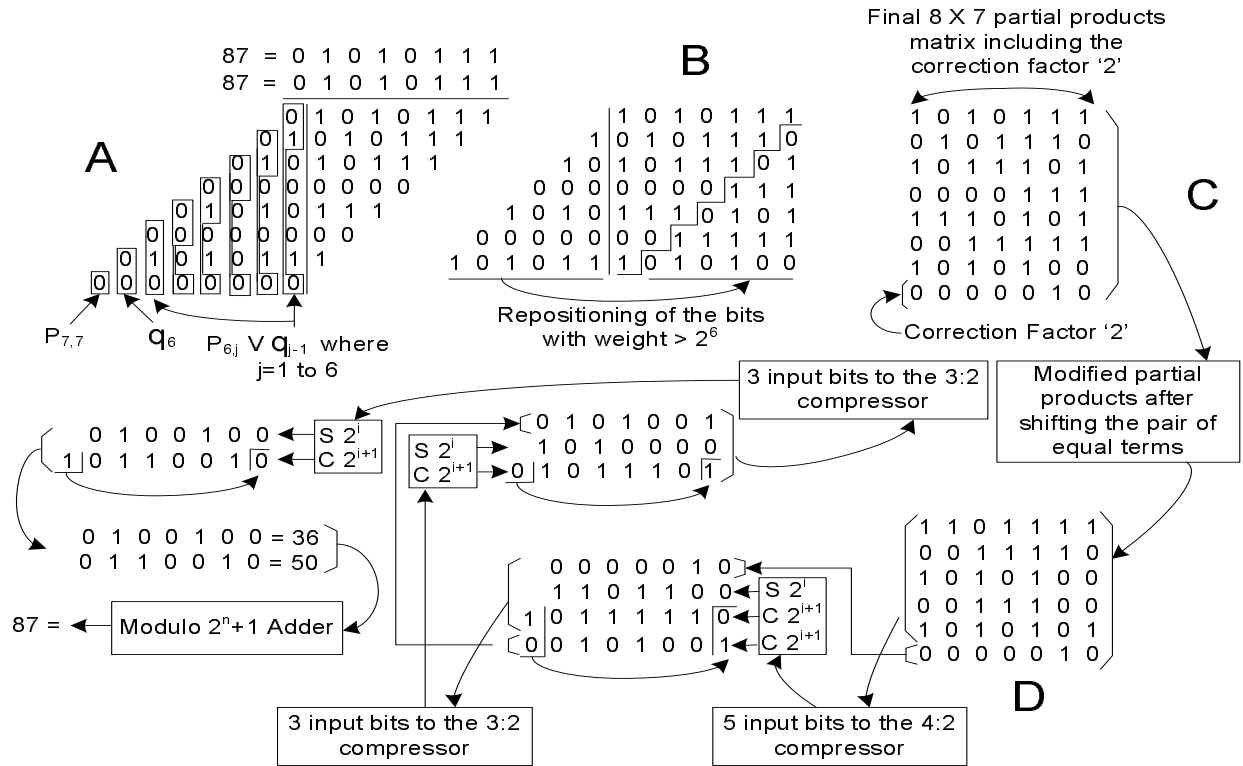


Fig. 8. An example of the proposed implementation of the modulo  $2^n + 1$  squarer

A. An example describing the partial products derivation, reduction and final output generation

Fig. 8 shows an example of the modulo  $2^n + 1$  squarer, which describes the partial products derivation, reduction and generation of the final output. 8-bit input  $001010111_2$  (i.e.,  $87_{10}$ ) is arbitrarily chosen and used to generate the result of  $01010111_2$  (i.e.,  $87_{10}$ ) in this example. In part A of Fig. 8, initial partial products generation is given where each bit of the last partial product vector ( $l_i$ ), the most significant bit of one partial product ( $m_i$ ) and penultimate bit of another partial product ( $u_i$ ) are ORed as shown in Fig. 8 (where  $l_i$ ,  $m_i$  and  $u_i$  are in the same column), part B shows the conversion of the initial partial products to  $7 \times 7$  partial product matrix. In this conversion, the bits with weight greater than  $2^6$  which are to the left of the straight line are complemented and repositioned to the right of the line, these repositioned bits are shown below the stair case line. In part C, the  $8 \times 7$  partial product matrix including the correction factor 2 is given. The pair of equal terms are shifted to the left column and the reduced partial products matrix is shown in part D. The 6 partial products are successively reduced to two final

vectors sum and carry using one 4:2 and two 3:2 compressors. In this process, the carry-outs with weight greater than  $2^6$  are repositioned to the rightmost column and are shown below the stair case line. These reduced final sum vector  $36_{10}$  ( $= 0100100_2$ ) and carry vector (the carry out bit is repositioned)  $50_{10}$  ( $= 0110010_2$ ) are modulo  $2^7 + 1$  added to produce the final result of  $87_{10}$  ( $= 01010111_2$ ).

#### IV. PROPOSED FAST LOW-POWER MODULO $2^n + 1$ MULTIPLIER/SQUARER IMPLEMENTATION

The hardware implementation of the modulo  $2^n + 1$  multiplier and squarer are similar in all aspects except the number of partial products is less in the squarer implementation. To present a clear and easy understanding of the proposed implementation, modulo  $2^n + 1$  squarer implementation is described. The proposed implementation of the mod  $2^n + 1$  squarer consists of three modules. The first module is to generate partial products, the second module is to reduce the partial products to two final operands and the last module is to add the Sum and Carry operands from partial products reduction module to compute the final result.

On comparing the proposed modulo squarer implementation with the existing modulo squarer implementations, the former implementation has several modifications and are tabulated below. Table I compares the existing implementation with the proposed one. Two  $n+1$  bit input operands

TABLE I  
DESIGN STEPS OF THE PROPOSED IMPLEMENTATION OF THE SQUARER VS. EXISTING METHOD.

	<b>Design steps</b>	<b>Existing implementation [20]</b>	<b>Proposed Implementation</b>
A	Number system	Weighted number system, no conversions are required	Diminished-1 number systems, conversions to and from weighted number system are required
B	Partial Products generation module	Involves complex logic functions to generate the partial product terms	Basic logic gates are used to generate the partial product terms
C	Partial products reduction module	Full Adders and Half Adders are used as the primitive blocks	Novel MUX-based compressors are used as the primitive blocks
D	Final stage addition module	Parallel prefix inverted EAC Adder is used	Sparse tree based Inverted EAC is used

are taken and partial products are generated using various logic functions for the existing and

the proposed methods. These partial products are easy to generate and the generation of the partial products has negligible effect on the critical path delay and the power consumed by the overall squarer. This module for modulo multiplier does not require any improvement, hence efficient and regular implementations from the literature are used without any changes. However, the partial products generation of the proposed modulo squarer is different from the existing implementation. The partial products reduction module takes the generated partial products and reduce them using carry save adder tree. However, the existing hardware implementation uses full adders to reduce these partial products and largely contributes to the critical path delay and power consumption. Therefore, these arrays of full adders in the partial product reduction module are to be replaced by the proposed MUX-based compressors which have less delay and power consumption compared to full adders. The final stage addition uses modulo  $2^n + 1$  adder to add Sum vector and Carry vector from the previous stage. In the parallel prefix based final stage adder implementations, inter-stage wiring and large carry merge cell density make the inverted End-Around-Carry (EAC) Kogge-Stone adder based on parallel prefix network an improper choice. A novel inverted EAC based on sparse tree adder is proposed and applied to reduce the complexity of inter-stage wiring and carry merge cell density in this work. In the following sections, detailed implementation of aforementioned three modules and an example of the proposed squarer implementation are presented and explained. These modules are explained for modulo  $2^n + 1$  multiplier, the same implementation is used for modulo squarers except the number of partial products in the latter implementation is less.

#### A. Partial products generation

From the above  $n \times n$  partial product matrix (shown in Fig. 6), we can observe that the partial product generation requires AND, OR and NOT gates. The most complex logic functions of partial product generation module are  $\overline{P_{i,j} \vee q_k}$  and  $P_{i,j} \vee q_k \vee P_{l,m}$ , where  $P_{i,j} = a_i b_j$  and  $q_k = P_{n,k} \vee P_{k,n}$ .

#### B. Partial products reduction

The partial product reduction unit is the most important module which mainly determines the critical path delay and the overall performance of the multiplier/squarer. Hence this module needs to be designed so as to get minimum delay and consume less power.

In the partial products reduction module, the  $n \times n$  partial product matrix and the constant 3 (i.e., correction factor) need to be added to produce the final sum and carry vectors. Zimmerman [16] demonstrated that  $(Sum + Carry + 1)$  modulo  $2^n + 1$  (final stage addition module) can also be calculated by  $(Sum + Carry)$  modulo  $2^n$  using inverted End-Around-Carry (EAC) adders, where  $Sum$  and  $Carry$  are  $n$ -bit vectors generated by the partial products reduction module. Modulo  $2^n$  inverted EAC adders have a regular structure that can be easily laid out for efficient VLSI implementation. Hence, instead of directly adding the correction factor of 3 to the  $n \times n$  partial product matrix in the partial products reduction module, an intermediate correction factor of 2 has to be added to the  $n \times n$  partial product matrix to save a constant 1 for the final stage addition module. Since there is a saved constant 1 available in the final stage addition module, modulo  $2^n$  inverted EAC adder can be used instead of the complex modulo  $2^n + 1$  adder.

Along with the constant 2, the  $n$  partial products should be added to produce the final  $n$ -bit Sum and Carry vectors. In a single stage of the Carry Save Addition, series of  $n$  full adders take 3 input operands and produce two  $n$ -bit output vectors. To add  $n + 1$  input operands,  $n - 1$  Carry Save Addition stages are required in the partial products reduction module. As the first partial product is the constant 2, in the first stage of the  $n - 1$  CSA stages, half adders can be used instead of full adders except for the second bit [20]. The  $n - 1$  stage CSA can be implemented using  $n$  full adders in each column of the  $n - 1$  stages [20]. In this regular implementation, series of full adders in the CSA adder columns can be replaced by the proposed MUX-based compressors that take the same number of inputs, which leads to more efficient implementation of the multiplier. For example, for a modulo  $2^8 + 1$  multiplier the existing CSA design uses 7 full adder stages in a single column to reduce the 9 partial products, the same reduction can be done by more efficient designs based on the proposed compressors. Two possible compressor-based implementations that can replace the existing design are shown in Fig. 9.

In the proposed compressor-based architecture, use of suggested compressors not only reduces the delay and power consumption but also the area of the circuit. For example, the full adder implementation requires fifteen full adders in series in any column for a modulo  $2^{16} + 1$  multiplier. However, these fifteen full adders can be replaced by two 7:2 compressors, one 5:2 compressor and two 3:2 compressors (Fig. 9).

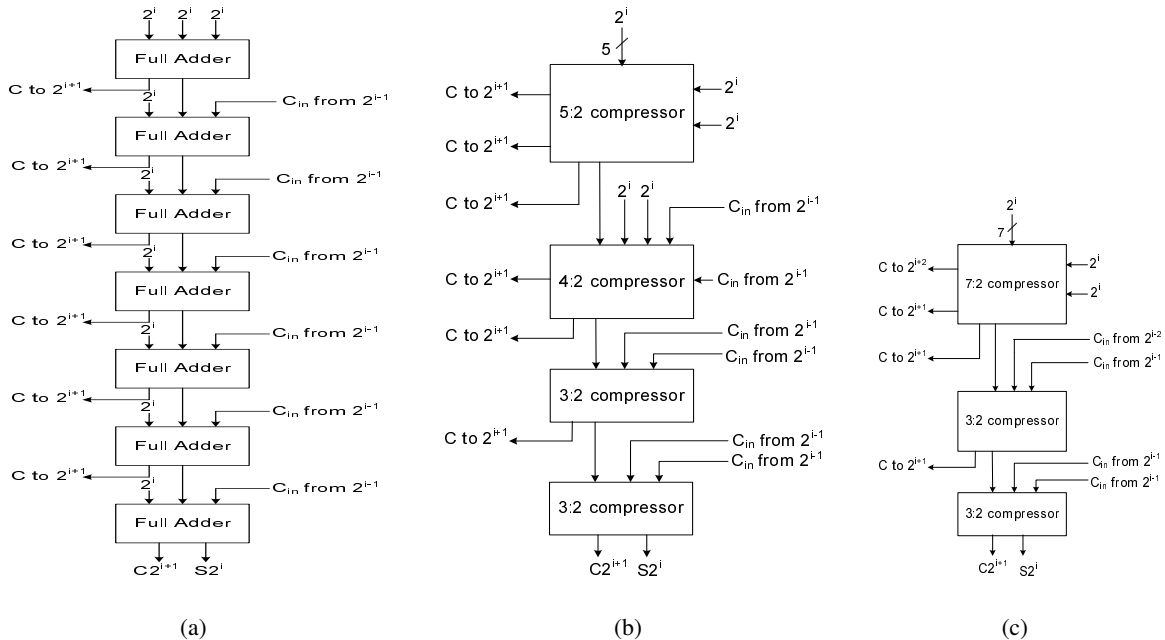


Fig. 9. Single column of the partial products reduction module of a modulo  $2^8 + 1$  multiplier (a) Using full adders (b) and (c) show two possible compressor based implementations. Implementation in (c) uses less number of compressors and results in more savings compared the implementation in (b)

### C. Compressors in each column of the partial products reduction module

The type of compressors in each column of the partial products reduction module varies with the value of  $n$ . In the proposed implementation of the multiplier/squarer, 7:2, 5:2, 4:2 and 3:2 compressors are used as the primitive blocks in the partial products reduction module. Compressors with maximum number of input bits result in higher savings in terms of power and delay factors. Hence, the order of preference for the compressors used in the implementation is 7:2 compressors, 5:2 compressors, 4:2 compressors and finally 3:2 compressors. Different implementations are possible with different combinations of the compressors. Best possible implementation for a multiplier can be obtained by using compressors with higher number of inputs. For example, the best possible configuration should contain one 7:2 compressor and two 3:2 compressors for  $n = 8$  case as shown in Fig. 9(c).

### D. Final Stage Addition

The partial products reduction module in the previous stage, generates one  $n$ -bit sum vector and one  $n$ -bit carry vector, which need to be added in the final stage addition module. As it is

a modulo  $2^n + 1$  multiplier/squarer, the Sum and Carry vectors should be modulo  $2^n + 1$  added. In the work of Zimmerman [16] it is shown that:

$$|Sum + Ccarry + 1|_{2^n+1} = |Sum + Carry + \overline{Cout}|_{2^n} \quad (7)$$

From (7) we can observe that the inverted carry out of the addition of Sum and Carry vectors has to be fed back. Hence, adding a constant '1' to the Sum and Carry vectors results in Inverted End Around Carry (EAC) modulo  $2^n$  addition which has regular VLSI implementation. The direct connection of the inverted carry out to the carry-in leads to oscillations in the circuit. Therefore, significantly efficient implementations of the modulo  $2^n$  Inverted End Around Carry adders are available in literature [16, 30]. The parallel prefix network based architecture of the modulo  $2^n$  Inverted EAC adder is the fastest known implementation [30].

The binary addition of two numbers using a parallel prefix network is done as follows. Let  $A = a_{n-1}a_{n-2} \dots a_1a_0$  and  $B = b_{n-1}b_{n-2} \dots b_1b_0$  be two weighted input operands to the network. The generate bit ( $g_i$ ) and propagate bit ( $p_i$ ) are defined as  $g_i = a_i \text{ AND } b_i$  and  $p_i = a_i \text{ OR } b_i$ , and these generate bits can be associated using the prefix operator  $\circ$  as follows:  $(g_i, p_i) \circ (g_{i-1}, p_{i-1}) = (g_i + p_i \cdot g_{i-1}, p_i \cdot p_{i-1}) = (g_{i:i-1}, p_{i:i-1})$  where  $+$  is the logical *OR* operator and  $\cdot$  is the logical *AND* operator.

The carry outs ( $C_i$ ) for all the bit positions can be obtained from the group generate ( $G_i = C_i$ ) where  $(G_i, P_i) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \dots \circ (g_1, p_1) \circ (g_0, p_0)$ . The parallel prefix network based Inverted EAC adder [30] achieves the addition of the input operands by recirculating the generate and the propagate bits at each existing level in  $\log_2 n$  stages. Let  $C_i^* (G_i^*)$  be the carry at bit position  $i$  in the Inverted EAC, this can be related to  $G_i$  as follows:

$$(G_i^*, P_i^*) = \begin{cases} \overline{(G_{n-1}, P_{n-1})} & \text{for } i = -1 \\ (G_i, P_i) \circ \overline{(G_{n-1:i+1}, P_{n-1:i+1})} & \text{for } n-2 \geq i \geq 0 \end{cases} \quad (8)$$

In the above equation  $\overline{(G_i, P_i)} = (\overline{G}, \overline{P})$ ,

where  $(G_i, P_i) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \dots \circ (g_1, p_1) \circ (g_0, p_0)$  and

$(G_{n-1:i+1}, P_{n-1:i+1}) = (g_{n-1}, p_{n-1}) \circ (g_{n-2}, p_{n-2}) \circ \dots \circ (g_{i+2}, p_{i+2}) \circ (g_{i+1}, p_{i+1})$ .

In some cases it is not possible to compute  $(G_i^*, P_i^*)$  in  $\log_2 n$  stages, then in these cases the equations in (8) are transformed into the equivalent ones as shown in Eq. (10) by using the following property [30]:

Suppose that  $(G^x, P^x) = (g, p) \circ \overline{(G, P)}$  and  $(G^y, P^y) = \overline{((\bar{p}, \bar{g}) \circ (G, P))}$

$$\begin{aligned} G^x &= g + p \cdot \bar{G} = \overline{\overline{g + p \cdot \bar{G}}} = \overline{\bar{g} \cdot (\bar{p} + G)} \\ &= \overline{(\bar{g} \cdot \bar{p} + \bar{g}G)} = \overline{(\bar{p} + \bar{g}G)} \end{aligned} \quad (9)$$

Therefore  $G^x = G^y$  and in (8)  $P^y$  is computed as  $p \cdot P$ .

To implement the parallel prefix computation efficiently, these transformations have to be applied  $j$  number of times recursively on  $(G_i, P_i) \circ \overline{(G_{n-1:i+1}, P_{n-1:i+1})}$  using the following relation:

$$n - 1 - i + j = \begin{cases} n, & \text{if } i > \frac{n}{2} - 1 \\ \frac{n}{2}, & \text{if } i \leq \frac{n}{2} - 1 \end{cases} \quad (10)$$

The new carry outs can be computed using the following equation:

$$(G_i^*, P_i^*) = \begin{cases} \overline{(G_{n-1}, P_{n-1})} & \text{for } i = -1 \\ \overline{(\bar{P}_i, \bar{G}_i) \circ (G_{n-1:i+1}, P_{n-1:i+1})} & \text{for } n - 2 \geq i \geq 0 \end{cases} \quad (11)$$

Hence, the transformations used above to achieve the parallel prefix computation in  $\log_2 n$  stages result in more number of carry merge cells and thereby adding more number of inter-stage wires. From the observation given in the previous section, parallel prefix adders suffer from excessive inter-stage wiring complexity and large number of cells, and these factors make parallel prefix based adders inefficient choices for VLSI implementations. Therefore, a new sparse tree based Inverted EAC adder is proposed in this work.

In sparse tree based Inverted EAC adders, instead of calculating the carry term  $G_i^*$  for each and every bit position, every  $K^{\text{th}}$  ( $K = 4, 8, \dots$ ) carry is computed. The value of  $K$  is chosen based on the sparseness of the tree, generally for 16 bit and 32-bit adders  $K$  is chosen as 4 [32]. The higher value of  $K$  results in higher value of non-critical path delay compared to critical path delay of  $O(\log_2 n)$  which should not be the case. The proposed implementation of the sparse tree based Inverted End-Around-Carry Adder (IEAC) is explained below clearly for 16-bit operands. For a 16-bit sparse IEAC with sparseness factor (i.e.,  $K$ ) equal to 4, the carries are computed for bit positions  $-1, 3, 7$  and  $11$ . Here, bit position  $-1$  corresponds to the inverted carry out  $\overline{(G_{15}, P_{15})}$  of the bit position 15. The carry out equations for the 16-bit Sparse tree IEAC are as follows:

$$\begin{aligned}
C_{-1}^* &= \overline{(G_{15}, P_{15})} = \overline{(g_1 5, p_1 5) \circ (g_{14}, p_{14}) \circ \dots \circ (g_1, p_1) \circ (g_0, p_0)} \\
C_3^* &= (G_3, P_3) \circ \overline{(G_{15:4}, P_{15:4})} = (g_3, p_3) \circ \dots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \dots \circ (g_5, p_5) \circ (g_4, p_4)} \\
C_7^* &= (G_7, P_7) \circ \overline{(G_{15:8}, P_{15:8})} = (g_7, p_7) \circ \dots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \dots \circ (g_9, p_9) \circ (g_8, p_8)} \\
C_{11}^* &= (G_{11}, P_{11}) \circ \overline{(G_{15:12}, P_{15:12})} = (g_{11}, p_{11}) \circ \dots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \dots \circ (g_{13}, p_{13}) \circ (g_{12}, p_{12})}
\end{aligned} \tag{12}$$

From (12) we can observe that some of the group generate terms need to be inverted to compute the final carries and also  $C_3^*$  cannot be computed in  $\log_2 n$  stages. Hence, using (9), (12) can be transformed and implemented in  $\log_2 n$  stages. Therefore, the new Carry out ( $C_i^*$ ) terms ( $G_i^*, P_i^*$ ) for a 16-bit Inverted EAC at every  $4^{th}$  bit can be formulated as follows:

$$\begin{aligned}
C_{-1}^* &= \overline{(G_{15}, P_{15})} = \overline{(g_1 5, p_1 5) \circ (g_{14}, p_{14}) \circ \dots \circ (g_1, p_1) \circ (g_0, p_0)} \\
C_3^* &= (G_3, P_3) \circ \overline{(G_{15:4}, P_{15:4})} = (g_3, p_3) \circ \dots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \dots \circ (g_5, p_5) \circ (g_4, p_4)} \\
&= \overline{(P_3, G_3)} \circ \overline{(G_{15:4}, P_{15:4})} \\
C_7^* &= (G_7, P_7) \circ \overline{(G_{15:8}, P_{15:8})} = (g_7, p_7) \circ \dots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \dots \circ (g_9, p_9) \circ (g_8, p_8)} \\
C_{11}^* &= (G_{11}, P_{11}) \circ \overline{(G_{15:12}, P_{15:12})} = (g_{11}, p_{11}) \circ \dots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \dots \circ (g_{13}, p_{13}) \circ (g_{12}, p_{12})}
\end{aligned}$$

Fig. 10 shows the finalized 16-bit sparse tree Inverted EAC adder. From Fig. 10, we can observe that all the carry outs are computed in  $\log_2 n$  stages with less number of carry merge cells and reduced inter-stage wiring intensity.

The Conditional Sum Generator (CSG) is implemented using ripple carry adder logic, two separate rails are run to calculate the carries  $C_{i+1}^*$ ,  $C_{i+2}^*$ ,  $C_{i+3}^*$  and  $C_{i+4}^*$  assuming the input carry  $C_i^*$  as 0 and 1. Four 2:1 multiplexers using the carry  $C_i^*$  from sparse tree network as 1-in-4

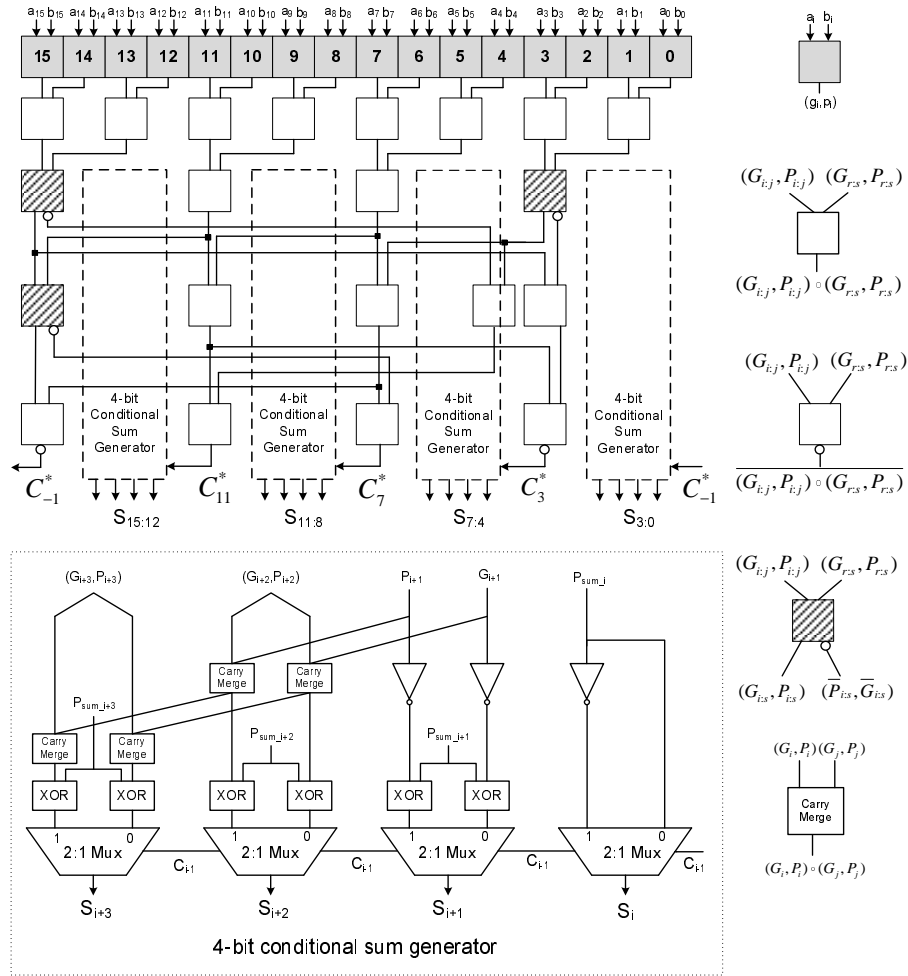


Fig. 10. 16-bit Sparse tree based Inverted EAC adder.

select line generate the final sum vector. The final sum is generated in  $\log_2 n$  stages in Inverted EAC parallel prefix adder and inverted EAC sparse tree adder with less number of cells and less inter-stage wiring in the proposed implementation. Hence, this approach results in low power and smaller area while providing better performance. The final result of the modulo  $2^n + 1$  multiplier has  $n + 1$  bits. The most significant bit is 1 only when  $Sum$  and  $Carry$  vectors of the Inverted EAC are complementary vectors which is equal to the Group propagate ( $P_{n-1}$ ) of this adder [20].

### E. An example of the proposed implementation

In this section, an example of the proposed implementation of the modulo  $2^n + 1$  squarer is given. The implementation of the modulo  $2^7 + 1$  squarer is given. Let  $X = x_n x_{n-1} \dots x_0$  be the input of the modulo squarer. The initial partial product matrix for this input is derived based on the partial product matrix shown in Fig. 7. The weights of the columns in the matrix range from  $2^0$  to  $2^7$ . The last partial product  $PP_6$  in the matrix is the total correction factor ( $COR_{Squarer}$ ). The initial partial product matrix of modulo  $2^n + 1$  squarer is shown in Fig. 11. From the above

	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
$PP_0$	$p_{6,0} \vee p_{7,6}$	$p_{5,0}$	$p_{4,0}$	$p_{3,0}$	$p_{2,0}$	$p_{1,0}$	$p_{0,0} \vee p_{7,6} \vee p_{7,7}$
$PP_1$	$p_{5,1}$	$p_{4,1}$	$p_{3,1}$	$p_{2,1}$	$p_{1,1}$	$p_{0,1}$	$p_{6,1} \vee p_{7,0}$
$PP_2$	$p_{4,2}$	$p_{3,2}$	$p_{2,2}$	$p_{1,2}$	$p_{0,2}$	$p_{6,2} \vee p_{7,1}$	$p_{5,2}$
$PP_3$	$p_{3,3}$	$p_{2,3}$	$p_{1,3}$	$p_{0,3}$	$p_{6,3} \vee p_{7,2}$	$p_{5,3}$	$p_{4,3}$
$PP_4$	$p_{2,4}$	$p_{1,4}$	$p_{0,4}$	$p_{6,4} \vee p_{7,3}$	$p_{5,4}$	$p_{4,4}$	$p_{3,4}$
$PP_5$	$p_{1,5}$	$p_{0,5}$	$p_{6,5} \vee p_{7,4}$	$p_{5,5}$	$p_{4,5}$	$p_{3,5}$	$p_{2,5}$
$PP_6$	$p_{0,6}$	$p_{6,6} \vee p_{7,5}$	$p_{5,6}$	$p_{4,6}$	$p_{3,6}$	$p_{2,6}$	$p_{1,6}$
$PP_7$	0	0	0	0	0	1	0

Fig. 11. Initial partial product matrix of the modulo  $2^7 + 1$  squarer

figure we can observe that two pair of terms are equal in each column. Even though, in the column with weight  $2^5$  there are three pairs of equal terms, we consider only two of those three pairs to shift to the left column. Three pairs can also be shifted to the left column, if there are any extra slots(i.e., zeros in the left column). The modified matrix consists of only six partial products including the correction factor and is shown in Fig. 12. The above set of partial products

	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
$PP_0$	$p_{6,0} \vee p_{7,6}$	$p_{4,0}$	$p_{3,0}$	$p_{2,0}$	$p_{1,0}$	$p_{5,2}$	$p_{0,0} \vee p_{7,6} \vee p_{7,7}$
$PP_1$	$p_{5,0}$	$p_{3,1}$	$p_{2,1}$	$p_{5,4}$	$p_{1,1}$	$p_{4,3}$	$p_{6,1} \vee p_{7,0}$
$PP_2$	$p_{4,1}$	$p_{3,2}$	$p_{2,2}$	$p_{6,4} \vee p_{7,3}$	$p_{5,3}$	$p_{6,2} \vee p_{7,1}$	$p_{1,6}$
$PP_3$	$p_{3,3}$	$p_{3,3}$	$p_{6,5} \vee p_{7,4}$	$p_{5,5}$	$p_{6,3} \vee p_{7,2}$	$p_{4,4}$	$p_{5,1}$
$PP_4$	$p_{0,6}$	$p_{6,6} \vee p_{7,5}$	$p_{5,6}$	$p_{4,6}$	$p_{3,6}$	$p_{2,6}$	$p_{4,2}$
$PP_5$	0	0	0	0	0	1	0

Fig. 12. Final partial product matrix of the modulo  $2^7 + 1$  squarer

are reduced into two final sum and carry vectors using a CSA network. This CSA network is

composed of efficient compressors instead of full adders and half adders. At each level of the CSA carry and sum bits are produced and are fed to the next subsequent level. The carry outputs at the leftmost column are fed back as carry inputs of the rightmost column of the CSA network. The sum vector and the carry vector produced by the partial products reduction module are driven to the final stage addition module (sparse tree based inverted end around carry adder) to generate the final output. The proposed implementation of modulo  $2^7 + 1$  squarer is shown in Fig. 13. From the Fig. 13, we can observe that the partial reduction module consists of one  $4 : 2$  and two  $3 : 2$  compressors in each column.

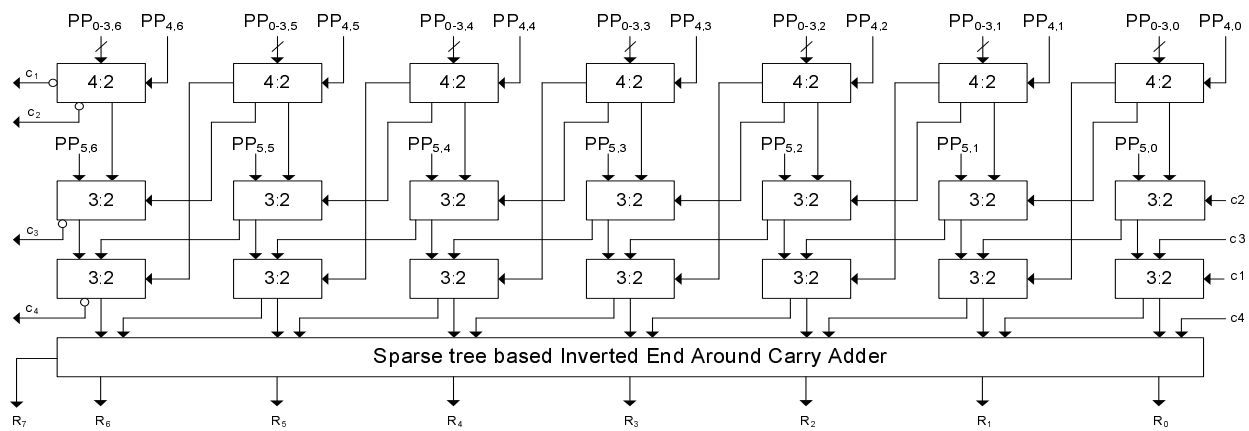


Fig. 13. Proposed implementation of the modulo  $2^7 + 1$  squarer using efficient compressors

## V. PARAMETRIC COMPARISON

Efficiency of the proposed modulo  $2^n + 1$  multiplier/squarer implementation is achieved by the newly-designed partial products reduction and final stage addition modules. The proposed modulo squarer has a new set of partial products compared to the existing implementation. As is shown in [20], which discusses the architectural characteristic comparisons of the various modulo  $2^n + 1$  multipliers, the multiplier proposed in [20] has been the most efficient implementation known in the literature so far. Hence, the proposed multiplier implementation is compared with the multiplier reported in [20]. The squarer implementation is compared with the known existing model. The comparisons are carried out using the unit-gate model proposed by Tyagi [35] and also experimental results are compared. The partial products reduction module largely contributes to the delay of the overall multiplier. For comparison purposes, the existing implementation

of the partial products reduction module is done using the Carry-Save-Adder (CSA) array implementation to reduce the interconnect delay and complexity. The delay terms of the proposed multiplier and squarer implementation consist of three parts as shown in (13).

$$\begin{aligned} T_{Proposed\_M} &= T_{PPG\_M} + T_{PPR\_M} + T_{FSA\_M} \\ T_{Proposed\_S} &= T_{PPG\_S} + T_{PPR\_S} + T_{FSA\_S} \end{aligned} \quad (13)$$

where  $PPG$ ,  $PPR$  and  $FSA$  represent the Partial Products Generation, Partial Products Reduction and Final Stage Addition modules, respectively.  $\_M$  and  $\_S$  notations indicate the multiplier and squarer parameters respectively.

The delay of the proposed multiplier/squarer consists of three parts. According to the unit-gate model shown in Tyagi [35], the delay of 2-input monotonic gates is 1-gate equivalent and delay of 2-input XOR and 2:1 MUX is 2 gate equivalents. In the partial products generation module, the most complex partial product term takes 3 unit-gate equivalents delay. Hence,

$$T_{PPG\_M} = T_{PPG\_S} = 3$$

The proposed partial products reduction module is composed of various compressors. For a different value of  $n$ , an efficient design is achieved by a formation of various compressors in the critical path. In the proposed implementation of the multiplier, only 7:2, 5:2, 4:2 and 3:2 compressors are used. For a 3:2 compressor, one XOR/XNOR and one MUX are used in the critical path, yielding a delay of 4 gate equivalents and 4:2 compressors have one XOR/XNOR and 2 MUX in the critical path with a total delay of 6 gate equivalents. Similarly, 5:2 and 7:2 compressors have 8 and 12 gates equivalent delays, respectively. Hence, the delay of the partial products reduction module can be formulated as:

$$T_{PPR\_M} = T_{PPR\_S} = 12 \cdot N(7 : 2) + 8 \cdot N(5 : 2) + 6 \cdot N(4 : 2) + 4 \cdot N(3 : 2) \quad (14)$$

In (14),  $N(i : 2)$ , where  $i=7, 5, 3$  and  $2$ , represents the number of  $(i : 2)$  compressors in the critical path. The FSA is implemented using sparse tree based Inverted EAC, the delay factor of FSA is divided into three factors including partial sum generation, carry propagate network, and MUX in the non-critical path. The partial sums can be generated using a 2-input XOR which has 2 unit-gate equivalents delay. The sparse carries are computed in  $\lceil \log_2 n \rceil$  levels. Each level is

an array of carry merge cells, and each carry merge cell has 2 gate equivalents delay. Therefore, the delay of the carry propagate network is  $2\lceil\log_2 n\rceil$  unit-gate equivalents. And finally, the sum bits are generated using two 2:1 MUX, with gate equivalent delay of 2. Thus, the delay of the FSA is:

$$T_{FSA\_M} = T_{FSA\_S} = 2 + 2\lceil\log_2 n\rceil + 2 = 2\lceil\log_2 n\rceil + 4$$

Therefore, the overall unit-gate equivalent delay of the proposed multiplier is:

$$\begin{aligned} T_{Proposed\_M} = T_{Proposed\_S} &= T_{PPG} + T_{PPR} + T_{FSA} \\ &= 3 + 12 \cdot N(7:2) + 8 \cdot N(5:2) + 6 \cdot N(4:2) + 4 \cdot N(3:2) + 2\lceil\log_2 n\rceil + 4 \\ &= 12 \cdot N(7:2) + 8 \cdot N(5:2) + 6 \cdot N(4:2) + 4 \cdot N(3:2) + 2\lceil\log_2 n\rceil + 7 \end{aligned}$$

In Table. II, the equivalent gate delays for the proposed implementations and well known existing implementations are compared for different values of  $n$ . It can be observed that the proposed multiplier and squarer perform considerably better as the value of  $n$  increases and the proposed implementations result in an average of 16% and 14% reduction in the overall delay of the multiplier and squarer respectively. The delay of the existing squarer does not include the delays of the weighted to/from diminished-1 number system.

TABLE II  
DELAY COMPARISONS FOR THE PROPOSED AND THE EXISTING IMPLEMENTATIONS

<b>n</b>	<b>4</b>	<b>8</b>	<b>12</b>	<b>16</b>	<b>20</b>	<b>24</b>	<b>28</b>	<b>32</b>
$T_{Proposed\_M}$	18	30	42	52	64	72	82	94
$T_{Vergos\_M}$	16	32	50	66	82	98	114	130
% Reduction_M	-12.5	6.25	16	21	21	26	28	27.6
$T_{Proposed\_S}$	21	29	35	41	47	51	57	61
$T_{Existing\_S}$	20	30	40	48	58	66	74	82
% Reduction_S	-5	3.3	12.5	14.5	18.9	22.7	22.9	25.6

Even though the unit gate model gives delay comparisons in terms of gate counts, the standard cell based implementation of the proposed compressor based multiplier/squarer gives

much more accurate delay and power estimations. The existing compressors and the proposed compressors have same gate counts, but the laid out implementation of the new compressor performs much better because of its MUX-based implementation. The actual power and delay factors are decided based on the inter-stage wires and complexity of the design. The parallel prefix network based Inverted EAC has a more number of inter-stage connections and intensified wiring complexity. Hence, for a better perspective for the sake of comparisons, the standard cell based implementations are designed and the power, delay comparisons are carried out. The proposed multipliers/squarers for various values of  $n$  are specified using Verilog Hardware Description Language (HDL). The verilog descriptions are mapped on a  $0.18 \mu\text{m}$  CMOS standard cell library using Leonardo Spectrum synthesis tool from Mentor Graphics. The design is optimized for high speed performance. Netlists generated from synthesis tool are passed on to standard route and place tool, the layouts are iteratively generated to get the circuits with minimum area. The proposed implementation and the carry save array based implementations presented in [20, 36] are implemented and compared with respect to power and delay. The obtained experimental results are shown in Table. III. The power values are given in  $mW$  and delay values are given in  $nS$ .

TABLE III  
EXPERIMENTAL RESULTS SHOWING AN AVERAGE REDUCTION OF 10%-12% IN POWER AND DELAY FACTORS

$n$	Proposed_M		Existing_M [20]		Proposed_S		Existing_S [36]	
	Power	Delay	Power	Delay	Power	Delay	Power	Delay
4	0.314	1.064	0.352	0.972	0.296	1.112	0.273	0.865
8	0.872	1.982	0.986	2.157	0.935	2.038	1.169	2.352
16	2.143	4.216	2.450	4.639	1.815	3.397	2.233	3.842
24	3.257	6.208	3.724	7.015	2.563	4.895	3.315	5.102
32	7.831	8.542	9.124	9.796	5.841	6.255	6.311	7.313

## VI. CONCLUSIONS

A novel implementation of the modulo  $2^n + 1$  multiplier and squarer are presented in this paper. The proposed design of the modulo  $2^n + 1$  multiplier/squarer uses compressors in the

partial products reduction stage, the use of the efficient compressors in place of full adders resulted in considerable improvements in terms of delay and power. A novel sparse tree based inverted EAC adder in the final stage addition, which has less wiring complexity and sparse carry merge cells compared to parallel prefix network based implementations. The proposed multiplier is compared with the most efficient modulo  $2^n + 1$  multiplier available in the literature. The unit gate model analysis and standard cell based implementations are carried out on the proposed implementation and the existing implementation to clearly demonstrate and verify the potential benefits from the proposed design. The proposed multiplier is proven to perform better than the existing implementation with respect to three major design criteria (i.e., power, delay and power-delay product). The proposed efficient designs can be applied to various applications requiring modulo  $2^n + 1$  operations such as the International Data Encryption Algorithm (IDEA) to achieve faster low-power operation while demanding less area.

#### REFERENCES

- [1] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, Eds., *Modern Applications of Residue Number System Arithmetic to Digital Signal Processing*. New York: IEEE Press, 1986.
- [2] W.K.Jenkins and B.J.Leon, "The use of residue number systems in the design of finite impulse response digital filters", *IEEE Trans.Circuits and Systems*, Vol.CAS-24, pp.191-201, April 1977.
- [3] Beckmann, P. E.; Musicus, B. R. Fast fault-tolerant digital convolution using a polynomial residue number system *IEEE Transactions on Signal Processing*, vol. 41, issue 7, pp. 2300-2313.
- [4] Leibowitz, LM.: A simplified binary arithmetic for the Fermat number transform, *IEEE Trans. Acoust. Speech Signal Process.*,1976, 24, pp. 35659
- [5] A. Curiger et. al., "VINCI: VLSI Implementation of the New Secret-keyBlock Cipher IDEA", *Proc. of the Custom Integrated Circuits Conference*, San Diego, USA, May 1993
- [6] Sklavos, N.; Koufopavlou, O., "Asynchronous low power VLSI implementation of the International Data Encryption Algorithm," *Electronics, Circuits and Systems*, 2001. ICECS 2001. The 8th IEEE International Conference on , vol.3, no., pp.1425-1428 vol.3, 2001.
- [7] Stefan Wolter, Holger Matz, Andreas Schubert, Rainer Laur: On the VLSI Implementation of the International Data Encryption Algorithm IDEA. *ISCAS 1995*:397-400
- [8] Zimmermann, R., Curiger, A., Bonnenberg, H., Kaeslin, H., Felber,N., and Fichtner, W. "A 177 Mb/s VLSI implementation of the international data encryption algorithm", *IEEE J. Solid-State Circuits*, 1994, 29, (3), pp. 303-307
- [9] A. Ashurand, M. Ibrahim, and A. Aggoun, Novel RNS structures for the moduli set  $(2 \cdot 1; 2 ; 2 + 1)$  and their application to digital filter implementation, *J. Signal Process.*, vol. 46, pp. 33143, 1995.
- [10] D. Gallaher, F. Petry, and P. Srinivasan, The digital parallel method for fast RNS to weighted number system conversion for specific moduli  $(2k \cdot 1; 2k; 2k + 1)$ , *IEEE Trans. Circuits Syst.II*, vol. 44, pp. 537, Jan. 1997.
- [11] A. Curiger,H. Bonnenberg, and H. Kaeslin, Regular VLSI architectures for multiplication modulo  $(2 + 1)$ , *IEEE J. Solid-State Circuits*, vol.26, no. 7, pp. 99094, Jul. 1991.

- [12] Hiasat, A.A.: A memoryless mod(2n+1) residue multiplier, *Electron. Lett.*, 1992, 28, (3), pp. 31415
- [13] Wrzyszczyk, A., and Milford, D.: A new modulo 2a + 1 multiplier *Proc. Int. Conf. Computer Design (ICCD3)*, 1995, pp. 61417
- [14] Yutai Ma, "A Simplified Architecture for Modulo (2n + 1) Multiplication," *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 333-337, Mar. 1998,
- [15] I. Koren, *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
- [16] Zimmerman, R., "Efficient VLSI implementation of modulo  $(2^n \pm 1)$  addition and multiplication" *IEEE trans. Comput.*, 2002, 51, pp. 1389-1399.
- [17] Chaves, R., and Sousa, L.: Faster modulo  $2^n + 1$  multipliers without booth recoding *Proc. XX Conf. Design of Circuits and Integrated Systems (DCIS5)*, 2005
- [18] Efstathiou, C., Vergos, H.T., Dimitrakopoulos, G., and Nikolos, D., "Efficient diminished-1 modulo  $2^n + 1$  multipliers", *IEEE Trans. Comput.*, 2005, 54, pp. 491-496.
- [19] Zhongde Wang, Graham A. Jullien, William C. Miller., "An efficient tree architecture for modulo  $2^n + 1$  multiplication" *VLSI Signal Processing* 14(3): 241-248 (1996)
- [20] Vergos, H.T.; Efstathiou, C., "Design of efficient modulo  $2^n + 1$  multipliers", *IET Comput. Digit. Tech.*, 2007, 1, (1), pp. 49-57.
- [21] Yi-Jung Chen, Dyi-Rong Duh, Yungshian Sam Han: Improved Modulo (2n+1) Multiplier for IDEA. *J. Inf. Sci. Eng.* 23(3): 911-923 (2007)
- [22] R. Zimmermann and W. Fichtner., "Low-power logic styles: CMOS versus pass-transistor logic" *IEEE J. Solid- State Circuits*, vol. 32, pp. 1079-1090, July 1997.
- [23] Veeramachaneni, S.; Avinash, L.; Rajashekhar Reddy M; Srinivas, M.B., "Efficient Modulo  $(2^k \pm 1)$  Binary to Residue Converters System-on-Chip for Real-Time Applications" *The 6th International Workshop on Dec. 2006* pp.195 - 200.
- [24] C-H Chang, J Gu, MZhang "Ultra low-voltage lowpower CMOS 4-2 and 5-2 compressors for fast arithmetic circuits" *IEEE J. Circuits and Systems I*, Volume: 51, Issue: 10 pp: 1985- 1997,2004.
- [25] Rouholamini, M.; Kavehie, O.; Mirbaha, A.-P.; Jasbi, S.J.; Navi, K., "A New Design for 7:2 Compressors" *Computer Systems and Applications, 2007. AICCSA '07. IEEE/ACS International Conference on 13-16 May 2007*, pp:474 - 478.
- [26] P. 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, pp. 786-793, Aug 1973.
- [27] Andrew Beaumont-Smith, Cheng-Chew Lim, "Parallel Prefix Adder Design," *arith*,pp.0218, 15th IEEE Symposium on Computer Arithmetic (ARITH-15 '01), 2001
- [28] R. P. Brent, H. T. Kung., "A regular layout for parallel adders" *IEEE Tr. Comp.*, C-31(3):260-264, Mar. 1982.
- [29] J. Sklansky, "Conditional-sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 226-231, June 1960.
- [30] Vergos, H.T.; Efstathiou, C.; Nikolos, D., "Diminished-one modulo 2n+1 adder design," *Computers, IEEE Transactions on* , vol.51, no.12, pp. 1389-1399, Dec 2002
- [31] Harris, D., "A taxonomy of parallel prefix networks," *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol.2, no., pp. 2213-2217 Vol.2, 9-12 Nov. 2003
- [32] Mathew, S.; Anders, M.; Krishnamurthy, R.K.; Borkar, S.; "A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core," *In IEEE Journal of Solid-State Circuits*, Volume 38, Issue 5, May 2003 Page(s):689 - 695.
- [33] Grad, J. Stine, J.E. "A Multi-Mode Low-Energy Binary Adder," *Fortieth Asilomar Conference on Signals, Systems and Computers*, Oct. 29 2006-Nov. 1 2006 pp. 2065-2068.

- [34] Yan Sun; Dongyu Zheng; Minxuan Zhang; Shaoqing Li, "High Performance Low-Power Sparse-Tree Binary Adders," Solid-State and Integrated Circuit Technology, 2006. ICSICT '06. 8th International Conference on , vol., no., pp.1649-1651, 23-26 Oct 2006.
- [35] Tyagi, A.: A reduced-area scheme for carry-select adders IEEE Trans. Comput., 1993, 42, (10), pp. 1163-1170
- [36] Vergos, H.T.; Efstathiou, C., "Diminished-1 modulo  $2n+1$  squarer design," Computers and Digital Techniques, IEE Proceedings - , vol.152, no.5, pp. 561-566, 9 Sept. 2005
- [37] Piestrak, S.J., "Design of squarers modulo  $A$  with low-level pipelining," Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on , vol.49, no.1, pp.31-41, Jan 2002