# A Fast Low-Power Modulo $2^n + 1$ Multiplier

Rajashekhar Modugu, Nohpill Park, Yong-Bin Kim and Minsu Choi

**Abstract**

Modulo $2^n + 1$ multiplier is one of the critical components in applications in the area of digital signal processing, data encryption and residue arithmetic that demand high-speed and low-power operation. In this paper, an efficient hardware architecture of modulo $2^n + 1$ multiplier is proposed and validated to address the demand. The proposed modulo $2^n + 1$ multiplier has 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 uses novel compressor designs and sparse tree adders as primitive building blocks for fast low-power operation. The partial products reduction module is completely redesigned using the novel compressors and the final addition module is implemented using a new less complex sparse tree based inverted end-around-carry adder. The resulting modulo $2^n + 1$ multiplier is 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 similar hardware implementations making it a viable option for efficient designs.

**Index Terms**

Modulo multipliers, 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 like International Data Encryption Algorithm (IDEA) [5, 6, 7, 8] are frequently used for secured transmission of data. 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 Fermant 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 high 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, 18, 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.

Cruiger et al. [11] proposed three methods to implement the multiplier. In their first method, $(n+1)\times(n+1)$ bit multiplier is used and followed by two modulo $2^n$ adders to correct the carry-out. Thier second method is based on multiplication by modulo $2^n+1$ adders. In this method the multiplier is divided into two parts. One is carry save adder and the other is final carry select adder. This method adopted the diminished-1 transform to reduce the design complexity. In the third method, multiplier is implemented using bit-pair recoding scheme. Among aforementioned methods, the last two implementations are known as suitable for full-custom design. Similarly, in the work of Hiasat [12], the modulo multiplier is implemented using $(n+1)$-bit multiplier followed by adder and logic gates.

Later in the work of Wrzyszcz and Milford [13], the $(n+1)\times(n+1)$ partial product matrix 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. This multiplier has a series of three $n$-bit adders and multiplexors in the final stage and has considerably regular structure that is suitable for VLSI implementations.

To overcome the problem of $n+1$ bit input length, diminished-1 representation was used and applied on some of the implementations. For diminished-1 operand representation, bit-pair Booth recoding technique

has been used in [14] to reduce the number of partial products to approximately $n/2$ at the cost of additional Carry Save Adders(CSA). In the work of Ma [14], 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 can be applied on this multiplier to speed up the operation. The circuitry for the $2^n$ correction is an overhead for this multiplier. In the work of Wang et al. [18] the diminished-1 number representation is used and based on Wallace tree architecture. This multiplier does not need any conversion circuits from diminished-1 to binary and vice versa. This multiplier can be implemented in VLSI in similar manner to the $n \times n$-bit Wallace tree binary multiplier. The modulo $2^n + 1$ with both the ordinary and the diminished-1 representations is proposed in the work of Sousa and Chaves [17]. In this work, the multiplier implementation is based on the modified Booth recoding scheme to speed up the reduction of the partial products and the correction of the final stage addition results. This has zero handling property but it has extra circuits for correction computation. Based on the observations of Wrzyszcz and Milford [13], Vergos and Efstathiou et al. [20] proposed a modulo multiplier which has only one correction factor and 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 encoding scheme.

Even though the multipliers proposed in [20] have achieved considerable enhancements in terms of power consumption and delay, the hardware implementation of the multiplier requires a special attention. The critical path of the multiplier depends on the partial products reduction stage which uses a Carry Save Adder (CSA) and on the final stage addition. Hence, the efficient design of Carry Save adder and the final stage adder is of high need.

In order to address this need, a new efficient hardware implementation of the modulo $2^n + 1$ multiplier will be proposed and validated in this work. The proposed hardware architecture has the following major improvements to achieve fast low-power operation:

- An efficient design of the 7:2 compressor is given and novel compressors which are used in the partial products reduction module of the multiplier are described.
- The full adder implementation of the Carry Save Adder array (partial products reduction) is efficiently designed using the novel compressors to reduce the delay and power consumption of the multiplier.
- A new sparse tree based inverted End Around Carry (EAC) Adder, which has less inter-stage wiring and less number of carry merge cells is proposed. The use of the sparse tree adder in the final stage

addition results in very efficient design of the modulo $2^n + 1$ multiplier.

The rest of this manuscript is organized as follows. In section II, novel multiplexor based compressors, parallel prefix adders and sparse tree adder are reviewed. Section III presents the algorithm from the work of Vergos and Efstathiou [20] which is used to implement the multiplier. In section IV, the proposed hardware implementation of the modulo $2^n+1$ multiplier and novel sparse tree based inverted End Around Carry (EAC) adder are described. Qualitative and quantitative comparisons of the proposed multiplier with the existing well known implementations are given in section V. In section VII 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$\overline{Y}$+$\overline{X}$Y. The same XOR can be implemented using MUX with inputs $X, \overline{X}$ 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. This also reduces the total number of garbage outputs. Existing CMOS designs of 2:1 MUX and 2-input XOR are shown in Fig. 1 for comparison.

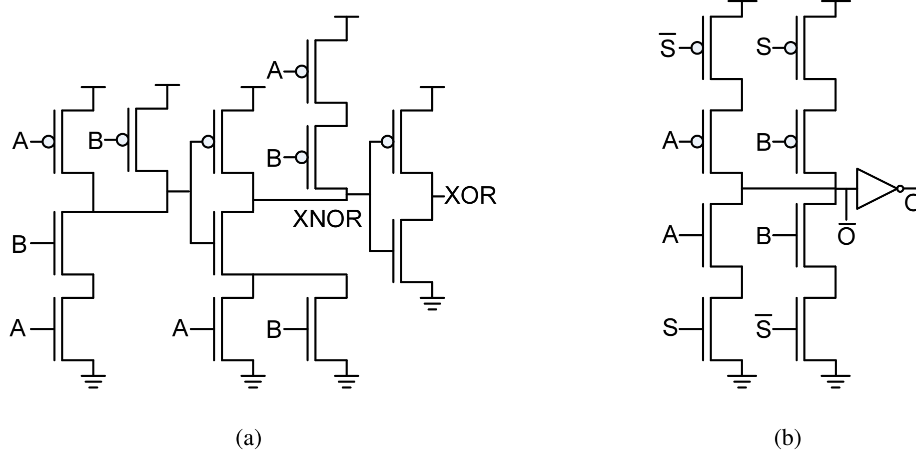


Fig. 1. CMOS implementation of 2-input (a) XOR (b) MUX

*2) Description of Compressors:* A $(p, 2)$ compressor has $p$ inputs $X_1, X_2 \ldots X_{p-1}, X_p$ 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:

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

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 these 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].

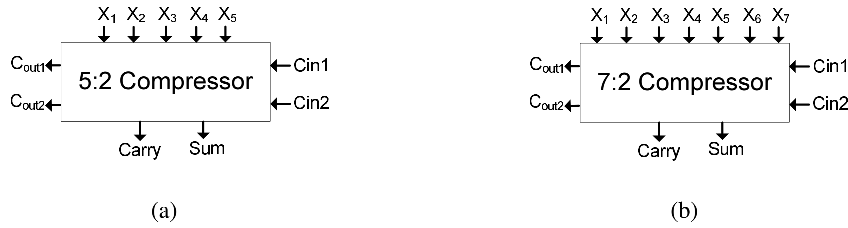


Fig. 2. Block diagrams of (a) 5:2 compressor (b) 7:2 compressor

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 designs of 5:2 [23] and 7:2 compressors are shown in Fig. 3, the delays of which are $\Delta(XOR) + 3\Delta(MUX)$ and $\Delta(XOR) + 5\Delta(MUX)$. CGEN block used in the non-critical path shown in this figure can be obtained from the equation $C_{outi} = (X + Y) \cdot Z + X \cdot Y$.

### B. Sparse Tree Adder

In binary addition operation, the critical path is determined by the carry computation module. Out of numerous formulations to design carry computation module, parallel prefix formulation [26] is delay

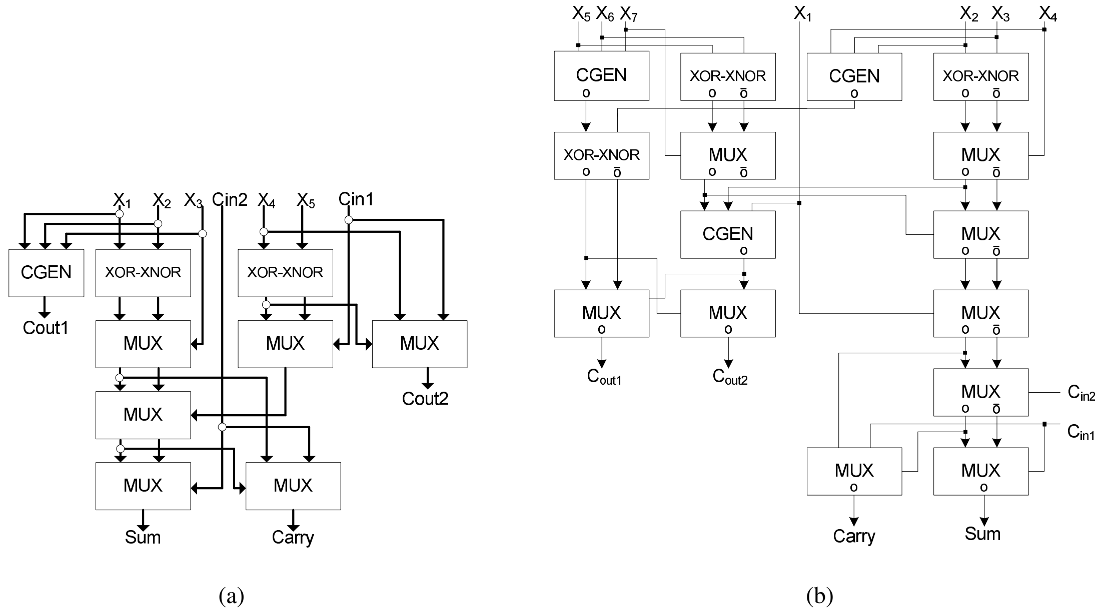


Fig. 3. Efficient designs of (a) 5:2 compressor [23] (b) proposed 7:2 compressor

effective and has regular structure suitable for efficient hardware implementation. In parallel prefix adders [26, 27], generate ($g_i$) and propagate ($p_i$) bits are circulated in $O(log_2n)$ stages to generate $n$ carries. The partial sum bits along with the carry bits produce the final sum vector. Parallel prefix adders have been realized in various attractive implementations such as Kogge-stone adders [26], Brent-kung [28] and Sklansky [29] to name a few. The performance of these adders varies with the logic depth, fan-outs and routing tracks [31]. Table I compares available adders in terms of logic depth, fan-out, number of carry merge cells and routing tracks. Of all these adders Kogge-stone adder [26] is known to be the fastest parallel prefix adder. However, the large number of carry merge cells and the complex inter stage wiring makes the hardware implementation of this adder extremely difficult.

TABLE I

COMPARISON OF THE VARIOUS PARALLEL PREFIX ADDERS

| **Adder type** | **Logic depth** | **Max fanout** | **Wiring tracks** | **# of Cells** |
|---|---|---|---|---|
| Kogge-Stone [26] | $log_2n$ | 2 | $n/2$ | $nlog_2n$ |
| Sklansky [29] | $log_2n$ | $n/2+1$ | 1 | $1/2log_2n$ |
| Brent-Kung [28] | $2log_2n-1$ | 2 | 1 | $2n$ |

An n-bit parallel prefix adder can be briefly explained 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 the $n$-bit input words to the prefix adder and the resulting sum vector be $F = f_{n-1}f_{n-2} \dots f_1f_0$. The common terms used in the carry computation of the prefix network are carry generate (i.e., $g_i$) and carry propagate (i.e., $p_i$) bits for bit position $i$ and are related to the bits $a_i$ and $b_i$ as $a_i$ AND $b_i$ and $a_i$ OR $b_i$, respectively. Successive carries can be obtained from these bits using the recursive formula $C_i = g_i + p_i \cdot C_{i-1}$ (where $+$ is an logical $OR$ operator and $\cdot$ is an logical $AND$ operator.)and the final sum is computed from partial sum term $h_i = a_i$ XOR $b_i$ and the carry out term $C_i$ using $s_i = h_i$ XOR $C_i$. In parallel prefix adders the carry computation is converted into prefix computation using the associative operator $\circ$ as

$$(g_m, p_m) \circ (g_n, p_n) = (g_m + p_m \cdot g_n, p_m \cdot p_n) \tag{1}$$

The carry term $C_i$ for bit position $i$, which is equivalent to $G_i$ can be calculated using the prefix operator $\circ$ as

$$(G_i, P_i) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}), \dots, (g_2, p_2) \circ (g_1, p_1) \circ (g_0, p_0) \tag{2}$$

The parallel prefix network achieves (2) by circulating generate and propagate bits in $O(log_2n)$ stages. Figure 4 shows the overall architecture of 16-bit Kogge-stone parallel prefix adder [26].

Performance of the parallel prefix adders is limited by the large number of carry merge cells and excessive inter-stage wiring tracks. The wiring delay and power consumption caused by routing tracks and carry merge cells make the prefix adders less attractive in practice. One possible solution is to reduce the number of carry merge cells and inter-stage wiring tracks. Hence, instead of computing all the carries, computing a subset of the carries can reduce the inter-stage wiring and carry merge cell density. This can be achieved by sparse tree adder architecture [32, 33], which has sparse carry computation block and conditional sum generator in critical and non-critical paths, respectively. It has been demonstrated in [32] that the proposed sparse carry computation results in 80% reduction of the wiring complexity which enables the use of wider wires on the carry generate and propagate signal paths that are most performance-critical in the design. Sparse tree adder functions similar to the parallel prefix adder except the carries are computed at every $4^{th}$ or $8^{th}$ bit position. Using these carries the final sum is generated using a conditional sum generator in the non-critical path. The conditional sum generator generates two different sums $Sum_i0$ and $Sum_i1$, assuming carries are computed by sparse carry computation block as zero and one, respectively. The 2:1 multiplexers which combine the critical and non-critical paths use the carries generated by sparse carry computation block to select the final sum. Various configurations

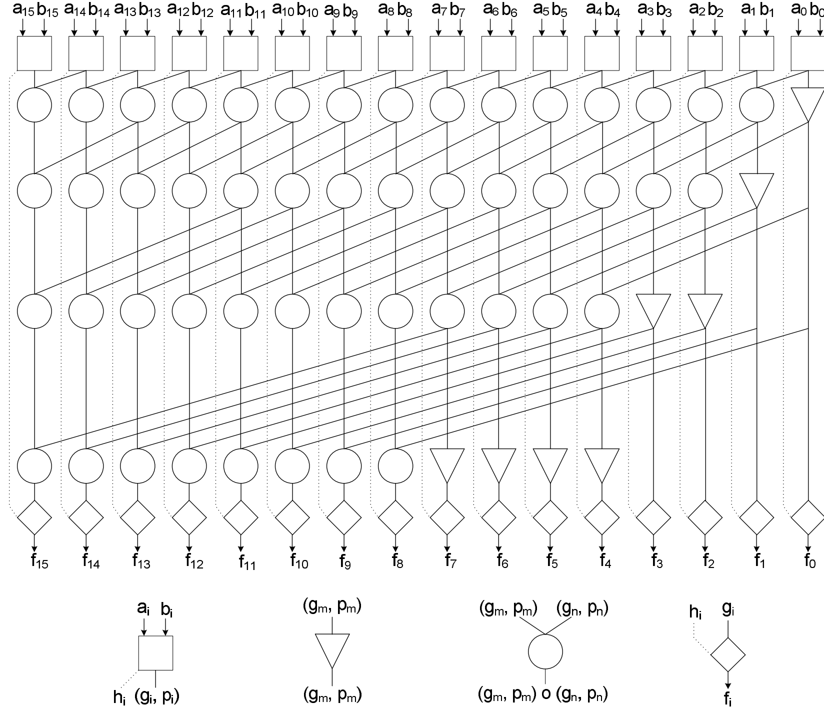


Fig. 4. 16-bit Kogge-Stone Parallel Prefix Adder [26]

[33, 34] of the sparse tree adder are possible by trading off logic depth, inter stage wiring and sparseness of the carry computation block. Sparse tree adders are high performance adders which consume lower power compared to prefix adders. A 16-bit sparse tree adder [32] with carry computed at every $4^{th}$-bit is shown in Fig. 5 for example.

## III. Algorithm for Implementation of Modulo $(2^n + 1)$ Multiplier

An algorithm for computation of $X \cdot Y \bmod (2^n + 1)$ is described below. 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. Hence, this algorithm is used for the proposed hardware implementation of the modulo multiplier. 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 mod $(2^n + 1)$ multiplication module takes two n-bit inputs and gives one n-bit output, by assigning the most significant bits of the inputs zeros and neglecting the most significant bit of the output.

Let $|A|_B$ denote the residue of $A$ modulo $B$. Let $X$ and $Y$ be two inputs represented as $X =$

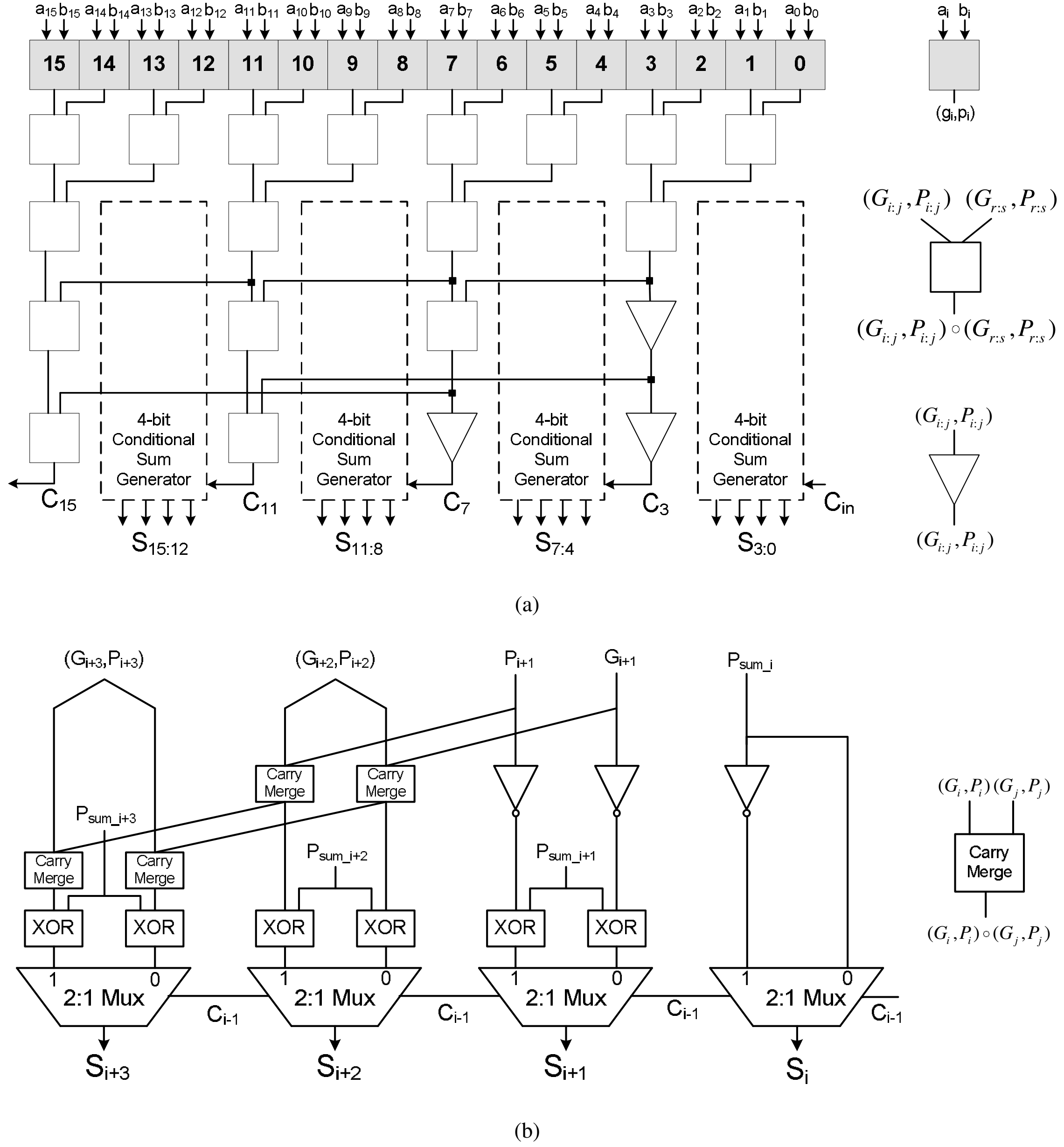


Fig. 5. 16-bit Sparse tree adder (a) Carry computation network in the critical path (b) 4-bit Conditional Sum Generator in the non-critical path

$x_n x_{n-1} \ldots x_0$ and $Y = y_n y_{n-1} \ldots y_0$ 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:

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

where $p_{i,j} = a_i$ AND $b_j$.

The $n \times n$ partial product matrix shown in Fig. 8 is derived from the initial partial product matrix in Fig. 6, 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. . .0,10. . .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. 7) 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 $p_{n,i}$ 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. 7.

| $2^{2n}$ | $2^{2n-1}$ | $2^{2n-2}$ | .... | $2^{n+2}$ | $2^{n+1}$ | $2^n$ | $2^{n-1}$ | $2^{n-2}$ | .... | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $p_{n,0}$ | $p_{n-1,0}$ | $p_{n-2,0}$ | .... | $p_{2,0}$ | $p_{1,0}$ | $p_{0,0}$ | |
| | | | | | $p_{n,1}$ | $p_{n-1,1}$ | $p_{n-2,1}$ | $p_{n-3,1}$ | .... | $p_{1,1}$ | $p_{0,1}$ | | |
| | | | B | $p_{n,2}$ | $p_{n-1,2}$ | $p_{n-2,2}$ | $p_{n-3,2}$ | $p_{n-4,2}$ | .... | $p_{0,2}$ | | | |
| | | | .... | .... | .... | .... | .... | .... | .... | | | | |
| | | $p_{n,n-2}$ | .... | $p_{4,n-1}$ | $p_{3,n-1}$ | $p_{2,n-1}$ | $p_{1,n-1}$ | $p_{0,n-2}$ | | | | | |
| C | $p_{n,n-1}$ | $p_{n-1,n-1}$ | .... | $p_{3,n-1}$ | $p_{2,n-1}$ | $p_{1,n-1}$ | $p_{0,n-1}$ | | | | | | A |
| $p_{n,n}$ | $p_{n-1,n}$ | $p_{n-2,n}$ | .... | $p_{2,n}$ | $p_{1,n}$ | $p_{0,n}$ | D | | | | | | |

Fig. 6. Initial partial product matrix

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

| $2^{2n-2}$ | .... | $2^{n+1}$ | $2^n$ | $2^{n-1}$ | $2^{n-2}$ | .... | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|---|
| | | | | $p_{n-1,0} \vee q_{n-1}$ | $p_{n-2,0}$ | .... | $p_{2,0}$ | $p_{1,0}$ | $p_{0,0} \vee p_{n,n} \vee q_{n-1}$ |
| | | | $p_{n-1,1} \vee q_0$ | $p_{n-2,1}$ | $p_{n-3,1}$ | .... | $p_{1,1}$ | $p_{0,1}$ | |
| | | $p_{n-1,2} \vee q_1$ | $p_{n-2,2}$ | $p_{n-3,2}$ | $p_{n-4,2}$ | .... | $p_{0,2}$ | | |
| | | .... | .... | .... | .... | .... | | | |
| | .... | $p_{3,n-2}$ | $p_{2,n-2}$ | $p_{1,n-2}$ | $p_{0,n-2}$ | | | | |
| $p_{n-1,n-1} \vee q_{n-2}$ | .... | $p_{2,n-1}$ | $p_{1,n-1}$ | $p_{0,n-1}$ | | | | | |

Fig. 7. Modified partial product matrix

$$\begin{aligned} \left|s2^i\right|_{2^n+1} &= \left|-s2^{|i|_n}\right|_{2^n+1} = \left|(2^n+1-s)2^{|i|_n}\right|_{2^n+1} \\ &= \left|\overline{s}2^{|i|_n} + 2^n 2^{|i|_n}\right|_{2^n+1} \end{aligned} \tag{3}$$

Equation (3) 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 2 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:

$$\begin{aligned} COR_1 &= 2^n\left[(2^1-1)+(2^2-1)+\ldots+(2^{n-1}-1)\right] \\ &= 2^n\left[2(1+2+2^2+\ldots+2^{n-2})-(n-1)\right] \\ &= 2^n(2^n-n-1) \end{aligned} \tag{4}$$

The $n \times n$ partial product matrix along with the equation (4) 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 (Sum vector and Carry vector). This can be done using a Carry Save Adder (CSA) of $(n-1)$ 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 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:

$$|c_i 2^n|_{2^n+1} = |-c_i|_{2^n+1} = |2^n + \overline{c_i}|_{2^n+1}$$

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

Fig. 8. Final n×n partial product matrix

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 in [20] produces $n-1$ such carry-out bits. Hence there is a second correction factor (5). The final correction factor because of the carry out bits of the CSA is:

$$COR_2 = |2^n(n-1)|_{2^n+1} \tag{5}$$

The final correction factor can be calculated from the sum of $COR_1$ and $COR_2$ as follows:

$$\begin{aligned} COR &= COR_1 + COR_2 \\ &= |2^n(n-1) + 2^n(2^n - n - 1)|_{2^n+1} \\ &= |2^n(2^n - 2)|_{2^n+1} = 3 \end{aligned} \tag{6}$$

The constant '3' in equation (6) is the final correction factor.

## IV. Proposed Implementation of the mod $2^n + 1$ Multiplier

The proposed implementation of the mod $2^n + 1$ multiplier 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. Even though the partial products generation module is easy to implement efficiently, for the sake of presenting the complete implementation of the multiplier, it is briefly described below.

Table II compares the existing implementation with the proposed one. Two $n+1$ bit input operands are taken and partial products are generated using various logic functions. 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 multiplier. This module does not require any improvement,

TABLE II

DESIGN STEPS OF THE PROPOSED IMPLEMENTATION OF THE MULTIPLIER VS. EXISTING METHOD.

| | Design steps | Existing implementation [20] | Proposed Implementation |
|---|---|---|---|
| A | Partial Products generation module | Uses basic logic gates to generate the partial products | Basic logic gates are used to generate the partial products |
| B | Partial products reduction module | Full Adders and Half Adders are the primitive blocks | Novel MUX-based compressors are the basic blocks |
| C | Final stage addition module | Parallel Prefix inverted EAC Adder is used | Sparse tree based Inverted EAC is used |

hence efficient and regular implementations from the literature are used without any changes. 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 $2^n + 1$ 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 new 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, an example of the proposed multiplier implementation and detailed implementation of aforementioned three modules are presented and explained.

### *A. An example of the proposed implementation*

Fig. 9 shows an example implementation of the proposed modulo $2^n + 1$ multiplier design. 9-bit multiplicand $001110111_2$ (i.e., $119_{10}$) and multiplier $001010111_2$ (i.e., $87_{10}$) are arbitrarily chosen and used to generate the result of $001001001_2$ (i.e., $73_{10}$) in this example. In part A of Fig. 9, 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. 9 (where $l_i$, $m_i$ and $u_i$ are in the same column), part B shows the conversion of the initial partial products

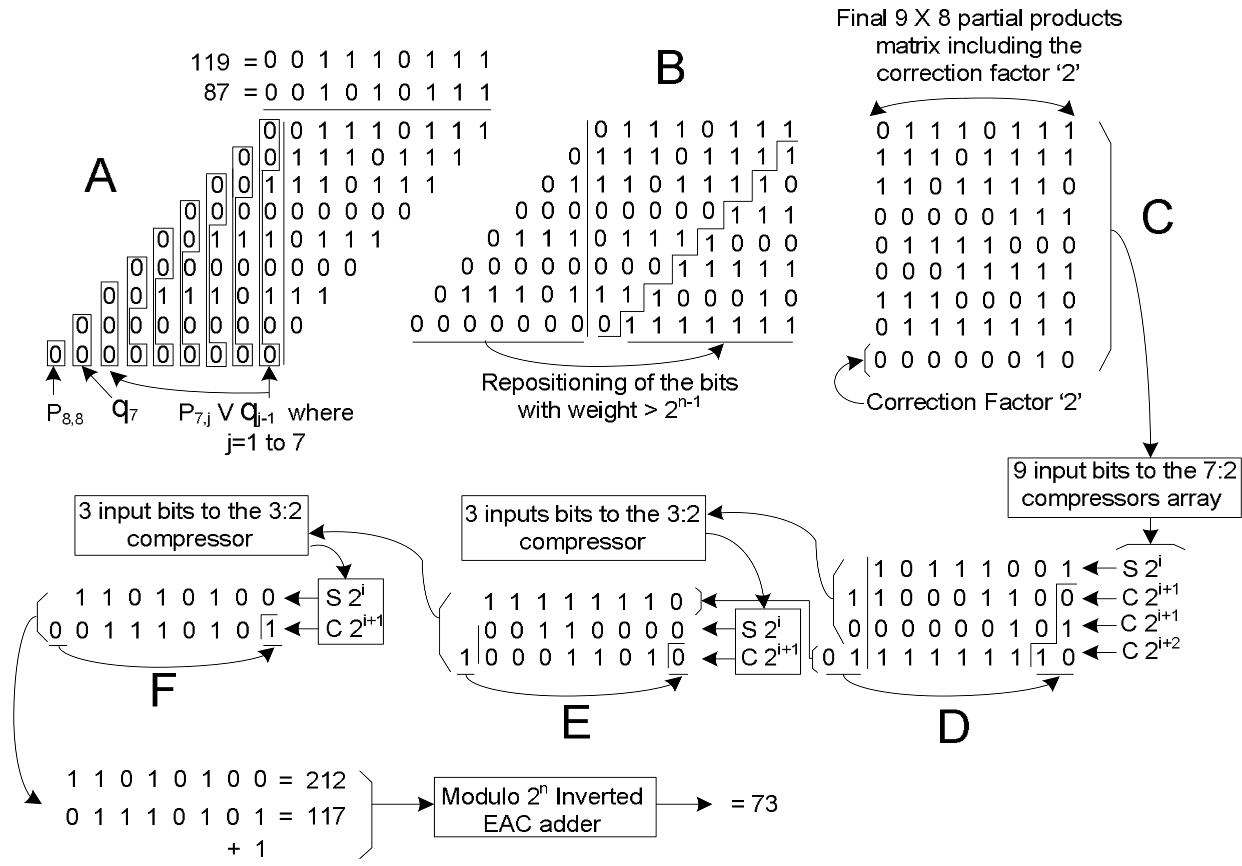


Fig. 9. An example of the proposed implementation of the multiplier

to $8 \times 8$ partial product matrix. In this conversion, the bits with weight greater than $2^7$ 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 final $9 \times 8$ partial product matrix including the correction factor 2 is given. The 9 partial products are reduced to four vectors $S2^i, C_1 2^{i+1}, C_2 2^{i+1}, C_3 2^{i+2}$ (where $i = 0 \ldots 7$), using a 7:2 compressor array as shown in part D. Again carry-outs with weight greater than $2^7$ are repositioned to the right and are shown below the stair case line. These four vectors are reduced to a final sum vector $212_{10}$ $(= 11010100_2)$ and carry vector (i.e., the carry out bit is repositioned) $117_{10}$ $(= 01110110_2)$ using two stages of 3:2 compressor arrays. Then, the modulo $2^8$ Inverted End Around Carry Adder adds the final sum and repositioned carry vector to produce the final result of $73_{10}$ $(= 001001001_2)$.

### B. Partial products generation

From the above $n \times n$ partial product matrix (shown in Fig. 8), 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 $q_k = P_{n,k} \vee P_{k,n}$. The complex gates of the partial product generation module are shown in Fig. 10.

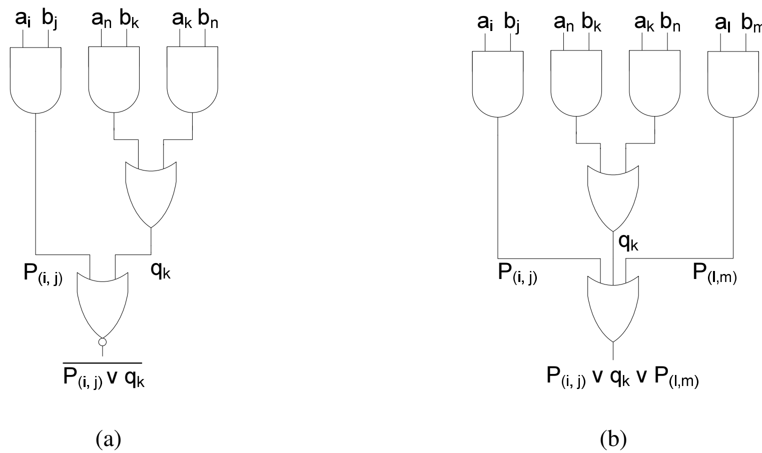


Fig. 10. Complex logic functions used to generate the partial product bits in the final partial product matrix shown in Fig. 8

### C. 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. 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 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 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 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 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. 11.

Computation of the correction factor COR for compressor implementation involves recomputing only $COR_2$, because $COR_1$ is obtained based on repositioning of the partial product terms, which is same for both implementations. The correction factor $COR_2$ computation for FA implementation which has $n-1$ stages of additions is shown in (III) and (5). And the $COR_2$ computation for the proposed multiplier implementation using the compressors also yields the same result.

For example, as shown in Fig. 9 the part C can be implemented using 7 full adders in all the columns producing a correction factor of 7. In parts D, E and F, the same partial product matrix is implemented using one 7:2 compressor and two 3:2 compressors producing a correction factors of 5, 1 and 1, respectively. Hence, $p$ bits with same weight $2^i$ can be added using $p-2$ FAs which give $p-2$ carry outs with weight $2^{i+1}$, resulting in a correction factor of $p-2$. The same $p$-bits can be added using a combination of compressors to produce a correction factor of $p-2$. Therefore, the overall correction factor COR computation for FA implementation and compressor implementation yield the same result of 3.

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.

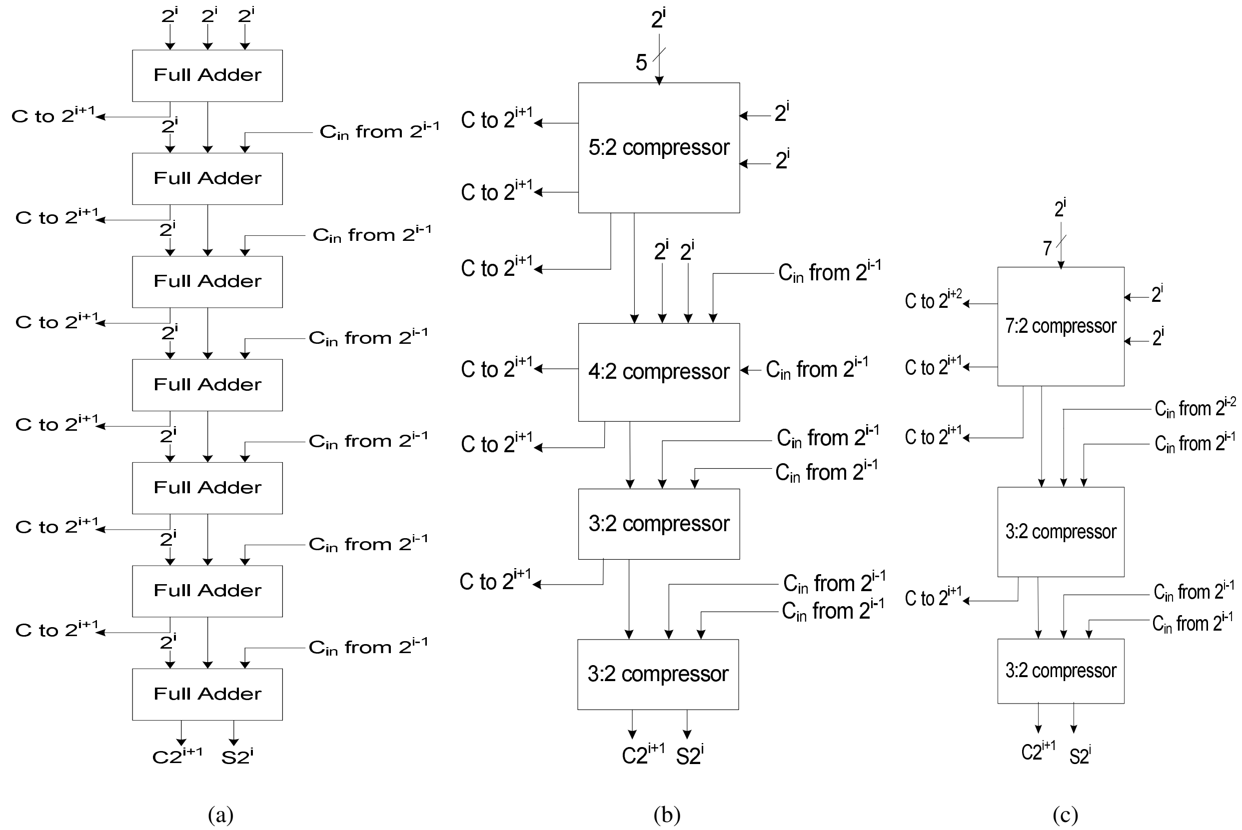


Fig. 11. Single column of the partial products reduction module of a modulo $2^8 + 1$ multiplier(a) Using full adders (b)and (c) show two possible compressor based implementations. Implementation in (c) uses less number of compressors and results in more savings compared the implementation in (b)

### *D. 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, 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 multiplier 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. In Fig. 11(c), for $n = 8$ the best possible combination contains one 7:2 compressor and two 3:2 compressors.

### *E. 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, 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 implementations. 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].

As it is explained in section II-B, the binary addition of two numbers using a parallel prefix network is done as follows. Let $A = a_{n-1}an - 2 \ldots a_1 a_0$ and $B = b_{n-1}bn - 2 \ldots b_1 b_0$ be two weighted input operands to the network. The generate bit $(g_i)$ and propagate bit $(p_i)$ are defined as $g_i = a_i$ AND $b_i$ and $p_i = a_i$ 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 an logical $OR$ operator and $\cdot$ is an 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 $(G_i, P_i) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \ldots (g_1, p_1) \circ (g_0, p_0)$. 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:

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

In the above equation $\overline{(G_i, P_i)} = (\overline{G}, P)$,
where $(G_i, P_i) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \ldots \circ (g_1, p_1) \circ (g_0, p_0)$ and
$(G_{n-1:i+1}, P_{n-1:i+1}) = (g_{n-1}, p_{n-1}) \circ (g_{n-2}, p_{n-2}) \circ \ldots \circ (g_{i+2}, p_{i+2}) \circ (g_{i+1}, p_{i+1})$.

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 (10) by using the following property [30]:

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

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

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

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:

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

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

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

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 Section II-B, 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 $K^{th}$ $(K = 4, 8 \ldots)$ 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 $\left(\overline{(G_{15}, P_{15})}\right)$ of the bit position 15. The carry out equations for the 16-bit Sparse tree IEAC are as follows:

$$\begin{aligned}
C^*_{-1} &= \overline{(G_{15}, P_{15})} = \overline{(g_15, p_15) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_1, p_1) \circ (g_0, p_0)} \\
C^*_3 &= (G_3, P_3) \circ \overline{(G_{15:4}, P_{15:4})} = (g_3, p_3) \circ \ldots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_5, p_5) \circ (g_4, p_4)} \\
C^*_7 &= (G_7, P_7) \circ \overline{(G_{15:8}, P_{15:8})} = (g_7, p_7) \circ \ldots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_9, p_9) \circ (g_8, p_8)} \\
C^*_{11} &= (G_{11}, P_{11}) \circ \overline{(G_{15:12}, P_{15:12})} = (g_{11}, p_{11}) \circ \ldots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_{13}, p_{13}) \circ (g_{12}, p_{12})}
\end{aligned} \tag{12}$$

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:

$$\begin{aligned}
C^*_{-1} &= \overline{(G_{15}, P_{15})} = \overline{(g_15, p_15) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_1, p_1) \circ (g_0, p_0)} \\
C^*_3 &= (G_3, P_3) \circ \overline{(G_{15:4}, P_{15:4})} = (g_3, p_3) \circ \ldots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_5, p_5) \circ (g_4, p_4)} \\
&= \overline{(\overline{P_3}, \overline{G_3}) \circ (G_{15:4}, P_{15:4})} \\
C^*_7 &= (G_7, P_7) \circ \overline{(G_{15:8}, P_{15:8})} = (g_7, p_7) \circ \ldots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_9, p_9) \circ (g_8, p_8)} \\
C^*_{11} &= (G_{11}, P_{11}) \circ \overline{(G_{15:12}, P_{15:12})} = (g_{11}, p_{11}) \circ \ldots \circ (g_0, p_0) \circ \\
&\quad \overline{(g_{15}, p_{15}) \circ (g_{14}, p_{14}) \circ \ldots \circ (g_{13}, p_{13}) \circ (g_{12}, p_{12})}
\end{aligned}$$

Fig. 12 shows a finalized 16-bit sparse tree Inverted EAC adder. From Fig. 12, we can observe that all the carry outs are computed in $log_2 n$ stages with less number of carry merge cells and less inter-stage wiring.

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 select line generate the final sum vector. The conditional sum generator is shown in Fig. II-B. The final sum is generated in $log_2 n$

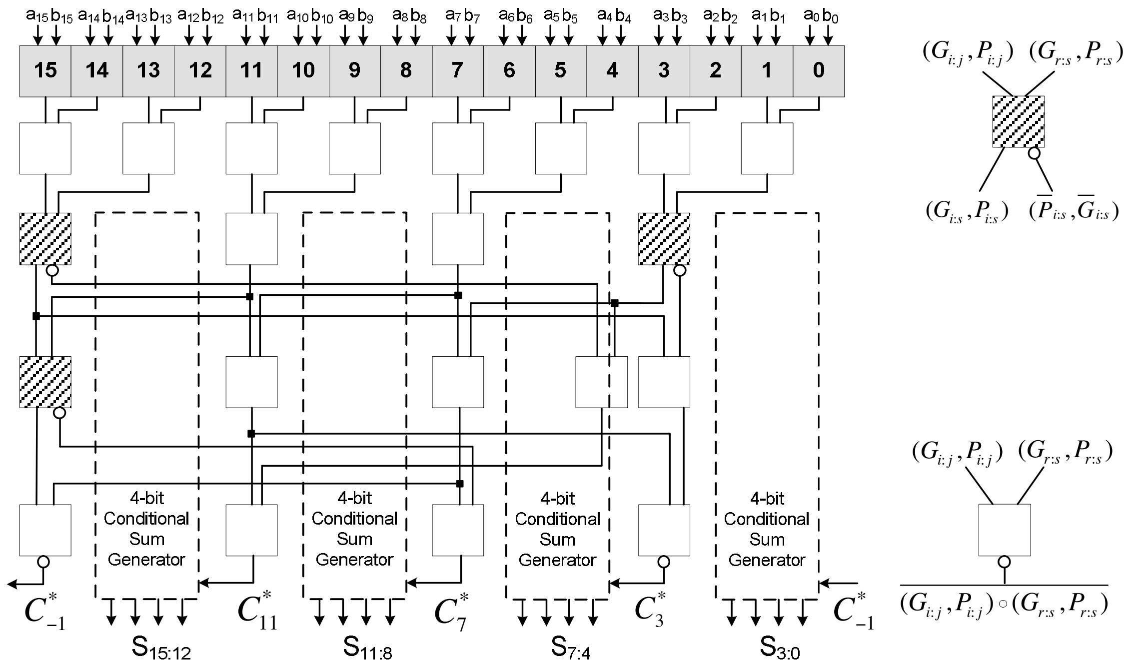


Fig. 12. 16-bit Sparse tree based Inverted EAC adder(4-bit Carry select adder is shown in Fig. II-B)

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]. Fig. 13 shows proposed implementation of the 17-bit modulo $2^{16}+1$ multiplier. In the Fig. 13 $PP_{i-j,k}$ represents bits of the final partial product matrix ( Fig. 8) from the rows $i$ to $j$ in the $k^{th}$ column and $R_{16}R_{15}\ldots R_2R_1R_0$ represents the final product of the modulo $2^{16}+1$ multiplier.

## V. Model-Based Parametric Comparison

The efficiency of the proposed multiplier implementation is achieved by the newly-designed partial products reduction and final stage addition modules. 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 comparisons are carried out using the unit-gate model proposed by Tyagi [35] and also experimental results are compared. The partial

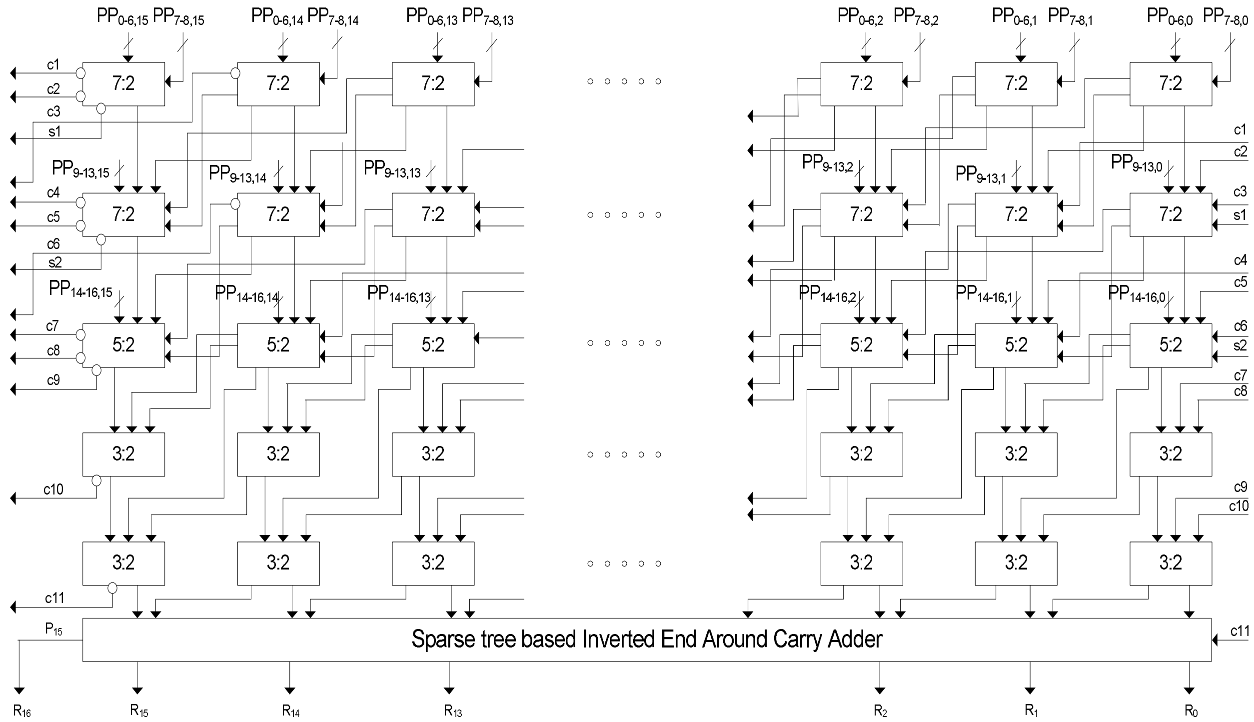


Fig. 13. Proposed implementation of the 17-bit input modulo $2^{16}+1$ multiplier($n = 16$)

products reduction module largely contributes to the delay of the overall multiplier. The implementation of the partial products reduction module is done using the Carry-Save-Adder (CSA) array implementation to reduce the interconnect delay and complexity. In the unit-gate model presented by Tyagi [35], each 2-input monotonic gate is considered as a single gate equivalent for both the area and delay comparisons, and the 2-input XOR gate and 2:1 MUX are considered as two gate equivalents (area and delay). The area and delay terms of the proposed multiplier implementation consist of three parts as shown in (13) and (14), respectively.

$$A_{Proposed} = A_{PPG} + A_{PPR} + A_{FSA} \tag{13}$$

$$T_{Proposed} = T_{PPG} + T_{PPR} + T_{FSA} \tag{14}$$

where $PPG$, $PPR$ and $FSA$ represent the Partial Products Generation, Partial Products Reduction and Final Stage Addition modules, respectively.

### A. Area Estimation based on Unit-Gate Model

The partial products generation module consists of implementing logic functions using primitive logic gates (i.e., AND, OR and NOT) [20]. The $n \times n$ partial products generation module uses $(n+1)^2$ AND (or NAND) gates, $2(n-1)$ OR (or NOR) gates and 4 more OR gates to generate the partial products. Hence, the estimated area of the PPG module using unit-gate model is:

$$A_{PPG} = (n+1)^2 + 2(n-1) + 4 = n^2 + 4n + 3$$

The proposed partial products reduction module is implemented using arrays of compressors. For different values of $n$, the efficient implementation can be achieved by different combinations of various compressors as shown in Fig. 11. The area of the partial products reduction module is the sum of the areas of the total compressors. The total number of 2-input XOR and 2:1 MUX in the proposed implementation of the 7:2, 5:2, 4:2 and 3:2 compressors are 15, 9, 6 and 3, respectively. Hence, the overall area of the 7:2, 5:2, 4:2 and 3:2 compressors is 30, 18, 12 and 6 unit gates, respectively. Thus, the gate equivalent area of the partial products reduction module is:

$$A_{PPR} = 30 \cdot n \cdot D(7:2) + 15 \cdot n \cdot D(5:2) + 12 \cdot n \cdot D(4:2) + 6 \cdot n \cdot D(3:2)$$

$D(i:2)$ in the above equation represents the number of $i:2$ compressors in the multiplier implementation. The sparse tree based Inverted EAC in the final stage addition module comprises critical and non-critical paths. The partial sum $(h_i)$, generate $(g_i)$ and propagate $(p_i)$ bits use $n$-XOR, $n$-AND and $n$-OR gates, respectively. The unit-gate model equivalent gate area is:

$$A_{PSGP} = 2n + n + n = 4n$$

where PSGP is the partial sum, generate and propagate bits generation section of the FSA. The critical path of the FSA consists of carry propagate network and the non-critical path consists of conditional sum generator. In the carry propagate network, carry is computed at every $4^{th}$-bit, hence in the first row of the network there are $\lceil \frac{n}{2} \rceil$ carry merge cells and in the remaining rows $(\lceil log_2 n \rceil)$ there are $\lceil \frac{n}{4} \rceil$ carry merge cells. As the carry is recirculated to form the inverted EAC, this brings an extra $\big(\lceil log_2 n \rceil - 3\big)$ carry merge cells in the last row. Equivalent gate area of the carry propagate network is:

$$A_{CPN} = 3\bigg(\lceil \frac{n}{2} \rceil + \big(\lceil log_2 n \rceil - 1\big)\lceil \frac{n}{4} \rceil + \big(\lceil log_2 n \rceil - 3\big)\bigg)$$

where CPN is the carry propagate network section of the FSA. Each carry merge cell outputs one group of generate $(g_{m:n} = g_m{\cdot}p_m{+}p_n)$ term of 2 gate equivalents area and one group of propagate $(p_{m:n} = p_m{\cdot}p_n)$ term of 1 gate equivalent area, resulting in a total of 3 gate equivalents of area. In the non-critical path, there are $\lceil \frac{n}{4} \rceil$ 4-bit conditional sum generators. As shown in Fig. 5(b), the conditional sum generator consists of four 2:1 MUXes, six XORs, four carry merge cells and three inverters. Therefore, the unit-gate equivalent area of the $\lceil \frac{n}{4} \rceil$ 4-bit conditional sum generator is $35 \cdot \lceil \frac{n}{4} \rceil$ gate equivalents. By summing all these, the area of the final stage addition module can be mathematically obtained as follows:

$$
\begin{aligned}
A_{FSA} &= A_{PSGP} + A_{CPN} + 35n/4 \\
&= 4n + 3\Big(\lceil \frac{n}{2} \rceil + \big(\lceil log_2 n \rceil - 1\big)\lceil \frac{n}{4} \rceil + \big(\lceil log_2 n \rceil - 3\big)\Big) + 35 \cdot \lceil \frac{n}{4} \rceil \\
&= 3\big(\lceil \frac{n}{4} \rceil + 1\big)\lceil log_2 n \rceil + 25 \cdot \lceil \frac{n}{2} \rceil - 9
\end{aligned}
$$

The overall gate equivalent area of the proposed multiplier using the unit-gate model is:

$$
\begin{aligned}
A_{Proposed} &= A_{PPG} + A_{PPR} + A_{FSA} \\
&= n^2 + 4n + 3 + A_{PPR} + 3\big(\lceil \frac{n}{4} \rceil + 1\big)\lceil log_2 n \rceil + 25 \cdot \lceil \frac{n}{2} \rceil - 9 \\
&= n^2 + A_{PPR} + 3\big(\lceil \frac{n}{4} \rceil + 1\big)\lceil log_2 n \rceil + 33 \cdot \lceil \frac{n}{2} \rceil - 6
\end{aligned}
$$

Besides, the gate equivalent area of the multiplier proposed in [20], using carry save adder array is:

$$
A_{Vergos} = 8n^2 + \frac{9}{2}n\lceil log_2 n \rceil - \frac{13}{2}n + 9
$$

In Table. III, the equivalent gate areas for the proposed implementation and CSA array implementation of the algorithm given in [20] are compared with respect to variable values of $n$ in $2^n + 1$. It can be observed that the proposed multiplier requires less area and results in 8% reduction in the overall area of the multiplier on average.

### *B. Delay Estimation based on Unit-Gate Model*

The delay of the proposed multiplier 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,

TABLE III

AREA COMPARISONS FOR THE PROPOSED AND THE EXISTING IMPLEMENTATIONS

| n | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|
| $A_{Proposed}$ | 160 | 553 | 1140 | 1968 | 3094 | 4383 | 5896 | 7825 |
| $A_{Vergos}$ | 147 | 577 | 1299 | 2241 | 3529 | 5001 | 6729 | 8713 |
| % Reduction | -8.84 | 4.15 | 12.24 | 12.18 | 12.32 | 12.35 | 12.37 | 10.19 |

$$T_{PPG} = 3$$

The proposed partial products reduction module is composed of various compressors. For a different value of $n$, an efficient design is achieved by a different 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:

$$T_{PPR} = 12 \cdot N(7:2) + 8 \cdot N(5:2) + 6 \cdot N(4:2) + 4 \cdot N(3:2) \tag{15}$$

In (15), $N(i:2)$, 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 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:

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

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

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

In Table. IV, the equivalent gate delays for the proposed implementation and CSA array implementation of the algorithm given in [20] are compared for different values of $n$. It can be observed that the proposed multiplier performs considerably better as the value of $n$ increases and the proposed implementation results in 8% reduction in the overall delay of the multiplier on average.

TABLE IV
DELAY COMPARISONS FOR THE PROPOSED AND THE EXISTING IMPLEMENTATIONS

| **n** | **4** | **8** | **12** | **16** | **20** | **24** | **28** | **32** |
|---|---|---|---|---|---|---|---|---|
| $T_{Proposed}$ | 18 | 30 | 42 | 52 | 64 | 72 | 82 | 94 |
| $T_{Vergos}$ | 16 | 32 | 50 | 66 | 82 | 98 | 114 | 130 |
| % Reduction | -12.5 | 6.25 | 16 | 21 | 21 | 26 | 28 | 27.6 |

## VI. EDA-BASED EXPERIMENTAL VERIFICATION OF THE PROPOSED DESIGN

Even though the unit gate model gives delay and area comparisons in terms of gate counts, the standard cell based implementation of the proposed compressor based multiplier gives much more accurate delay, area 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 more number of inter-stage connections and wire complexity. Hence, for a better perspective for the sake of comparisons, the standard cell based implementations are designed and the area, power and delay comparisons have been carried out using Mentor Graphics Electronic Design Automation (EDA) suite.

The proposed multipliers for $4 \leq n \leq 32$ (i.e., $n$ is word size) are specified using Verilog Hardware Description Language (HDL). The multiplier descriptions are mapped on a $0.18 \mu m$ CMOS standard cell library using Leonardo Spectrum synthesis tool from Mentor Graphics. The design is optimized for high speed performance at typical operating conditions (i.e., 1.8 V and 25°C). Netlists generated from

TABLE V

EXPERIMENTAL RESULTS SHOWING AREA (IN $mm^2$). THE LAST TWO COLUMNS, $\%_{min}$ AND $\%_{max}$, REPRESENT MINIMUM PERCENTAGE REDUCTION AND THE MAXIMUM PERCENTAGE REDUCTION IN AREA ACHIEVED BY THE PROPOSED DESIGN OVER COMPARED EXISTING DESIGNS.

| $n$ | Wang et al. [18] | Efstanthiou et al. [19] | Vergos et al. [20] | Proposed | $\%_{min}$ | $\%_{max}$ |
|---|---|---|---|---|---|---|
| 4 | 3.739 | 3.214 | 3.317 | 3.042 | 8.29% | 18.642% |
| 8 | 10.427 | 9.335 | 9.129 | 8.396 | 8.03% | 19.48% |
| 16 | 39.231 | 35.560 | 33.239 | 30.336 | 8.73% | 22.67% |
| 24 | 85.742 | 79.237 | 73.426 | 68.725 | 6.4% | 19.85% |
| 32 | 136.219 | 130.953 | 126.642 | 112.619 | 11.07% | 17.32% |

TABLE VI

EXPERIMENTAL RESULTS SHOWING POWER DISSIPATION (IN $mW$). THE LAST TWO COLUMNS, $\%_{min}$ AND $\%_{max}$, REPRESENT MINIMUM PERCENTAGE REDUCTION AND THE MAXIMUM PERCENTAGE REDUCTION IN POWER DISSIPATION ACHIEVED BY THE PROPOSED DESIGN OVER COMPARED EXISTING DESIGNS.

| $n$ | Wang et al. [18] | Efstanthiou et al. [19] | Vergos et al. [20] | Proposed | $\%_{min}$ | $\%_{max}$ |
|---|---|---|---|---|---|---|
| 4 | 0.94 | 0.86 | 0.71 | 0.73 | 22.34% | −2.81% |
| 8 | 3.33 | 2.76 | 2.63 | 2.47 | 40.48% | 6.08% |
| 16 | 9.18 | 7.71 | 7.24 | 6.82 | 25.7% | 5.8% |
| 24 | 12.39 | 11.12 | 10.87 | 9.36 | 24.45% | 13.89% |
| 32 | 18.32 | 16.96 | 14.02 | 13.32 | 27.29% | 4.99% |

synthesis tool are passed onto standard place and route tool, the layouts are iteratively generated to get the circuits with minimum area. The proposed implementation and modular multiplier implementations presented in [18, 19, 20] are implemented and compared with respect to area, power and delay which are commonly used design criteria in optimizing VLSI systems.

Table V shows EDA-based simulation results for area comparison; the proposed design vs. the known-best existing ones. For each $n$ (i.e., word size), the minimum percentage reduction (i.e., $\%_{min}$) and maximum percentage reduction (i.e., $\%_{max}$) in area are shown as well. As shown in table, the proposed

TABLE VII

EXPERIMENTAL RESULTS SHOWING DELAY (IN $ns$). THE LAST TWO COLUMNS, $\%_{min}$ AND $\%_{max}$, REPRESENT MINIMUM PERCENTAGE REDUCTION AND THE MAXIMUM PERCENTAGE REDUCTION IN DELAY ACHIEVED BY THE PROPOSED DESIGN OVER COMPARED EXISTING DESIGNS.

| $n$ | Wang et al. [18] | Efstanthiou et al. [19] | Vergos et al. [20] | Proposed | $\%_{min}$ | $\%_{max}$ |
|---|---|---|---|---|---|---|
| 4 | 1.52 | 1.41 | 1.08 | 1.09 | 28.28% | −0.92% |
| 8 | 2.27 | 2.03 | 1.87 | 1.72 | 24.23% | 8.02% |
| 16 | 2.86 | 2.69 | 2.58 | 2.49 | 12.93% | 3.49% |
| 24 | 3.15 | 3.11 | 3.01 | 2.92 | 7.3% | 2.99% |
| 32 | 3.92 | 3.71 | 3.65 | 3.56 | 9.18% | 2.47% |

deign achieves significant reduction in area (i.e., ranging from 6.4% to 22.67%). The proposed MUX-based compressor-based optimized compressor network design and Inverted EAC mainly contribute in improvement in area by significantly reducing the overall transistor count. EDA simulation results shown in this table are also in accordance with the results obtained from the unit-gate model in the previous chapter.

Power dissipation of the proposed design is also compared with the other existing designs in Table VI. Other than one case where $\%_{max}$ (i.e., maximum percentage reduction in power dissipation) is $-2.81\%$ for $n = 4$, the proposed design achieves substantial reduction in power dissipation over the existing designs ranging from 4.99% to 40.48%. In addition to reduced transistor count due to efficient compressors and compressor network, Inverted EAC used in the proposed design minimizes the overall wiring complexity of the adders and therefore saves unnecessary energy consumption.

Lastly, delay of the proposed design is compared with the existing methods in Table VII. Once again, there is one case where the proposed design is not better than one of the existing design and $\%_{max} = -0.92\%$ for $n = 4$ case). Other than that, the proposed design provides considerable reduction in delay over the existing designs ranging from 2.47% to 28.28%. In the proposed design, efficient compressors are automatically selected to form compressor columns with minimum critical-path gate delays in partial products reduction module. This compressor network optimization is the primary source of the observed reduction in delay over the existing designs.

## VII. CONCLUSIONS

An efficient implementation of the modulo $2^n + 1$ multiplier is presented in this paper. The proposed novel implementation takes advantage of newly-designed efficient compressors and sparse tree based Inverted End-Around-Carry (EAC) adders. The proposed design of the modulo $2^n + 1$ multiplier 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. An efficient 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 implementations available in the literature. The unit gate model analysis and EDA-based parametric simulation are carried out on the proposed implementation and the existing implementations to clearly demonstrate and verify potential benefits from the proposed design. The proposed multiplier is verified to outperform the existing implementations with respect to three major design criteria (i.e., area, power and delay).

## 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, No. 7, pp. 2300-2313, Jul 1993.

[4] Leibowitz, LM.: "A simplified binary arithmetic for the Fermat number transform," IEEE Trans. Acoust. Speech Signal Process., Vol. 24, pp. 356-359, 1976.

[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. and Koufopavlou, O., "Asynchronous Low Power VLSI Implementation of the InternationalData Encryption Algorithm," IEEE International Conference on Electronics, Circuits and Systems, Vol. 3, pp. 1425-1428, September 2-5, 2001.

[7] Stefan Wolter, Holger Matz, Andreas Schubert, Rainer Laur: "On the VLSI Implementation of the International Data Encryption Algorithm IDEA," IEEE International Conference on Electronics, Circuits and Systems, pp. 397-400, 1995.

[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, Vol. 29, pp. 303-307, 1994.

[9] A. Ashurand, M. Ibrahim, and A. Aggoun, Novel RNS structures for the moduli set $(2n - 1;\ 2n\ ;\ 2n + 1)$ and their application to digital filter implementation, J. Signal Process., Vol. 46, pp. 331-343, 1995.

[10] D. Gallaher, F. Petry, and P. Srinivasan, "The digital parallel method for fast RNS to weighted number system conversion for specific moduli $(2k - 1;\ 2k;\ 2k + 1)$, IEEE Trans. Circuits Syst. II, vol. 44, pp. 53-67, Jan 1997.

[11] A. Curiger,H. Bonnenberg, and H. Kaeslin, ”Regular VLSI architectures for multiplication modulo $(2^n + 1)$, IEEE J. Solid-State Circuits, Vol.26, No. 7, pp. 990-994, Jul 1991.

[12] Hiasat, A.A.: A memoryless mod$(2^n + 1)$ residue multiplier, Electron. Lett., Vol. 28, No. 3, pp. 314-315, 1992.

[13] Wrzyszcz, A., and Milford, D.: ”A new modulo $2^n + 1$ multiplier,” Proc. Int. Conf. Computer Design, pp. 614-617, 1995

[14] Yutai Ma, ”A Simplified Architecture for Modulo $(2^n + 1)$ 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., Vol. 51, pp. 1389-1399, 2002.

[17] Chaves, R., and Sousa, L.: ”Faster modulo $2^n + 1$ multipliers without booth recoding,” Conf. Design of Circuits and Integrated Systems, 2005.

[18] Zhongde Wang, Graham A. Jullien and William C. Miller., ”An efficient tree architecture for modulo $2^n+1$ multiplication,” VLSI Signal Processing, Vol. 14, No. 3, pp. 241-248, 1996.

[19] Efstathiou, C., Vergos, H.T., Dimitrakopoulos, G., and Nikolos, D., ”Efficient diminished-1 modulo $2^n + 1$ multipliers,” IEEE Trans. Comput., Vol. 54, pp. 491-496, 2005.

[20] Vergos, H.T. and Efstathiou, C., ”Design of efficient modulo $2^n + 1$ multipliers”, IET Comput. Digit. Tech., Vol. 1, No. 1, pp. 49-57, 2007.

[21] Yi-Jung Chen, Dyi-Rong Duh and Yunghsian Sam Han, ”Improved Modulo $(2^n + 1)$ Multiplier for IDEA,” J. Inf. Sci. Eng., Vol. 23, No. 3, pp. 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 and 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 Soc for Real-Time Applications, pp. 195-200, Dec 2006.

[24] C-H. Chang, J. Gu and M. Zhang, ”Ultra low-voltage low-power CMOS 4-2 and 5-2 compressors for fast arithmetic circuits,” IEEE J. Circuits and Systems I, Vol. 51, No. 10, pp. 1985-1997, 2004

[25] Rouholamini, M., Kavehie, O., Mirbaha, A.-P., Jasbi, S.J. and Navi, K., ”A New Design for 7:2 Compressors,” IEEE/ACS International Conference on Computer Systems and Applications, pp. 474-478, 2007.

[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 and Cheng-Chew Lim, ”Parallel Prefix Adder Design,” IEEE Symposium on Computer Arithmetic, pp. 218, 2001

[28] R. P. Brent and H. T. Kung., ”A regular layout for parallel adders” IEEE Tr. Comp., Vol. C-31, No. 3, pp. 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. and Nikolos, D., ”Diminished-one modulo $2^n + 1$ adder design,” IEEE Trans. on Comp., 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, pp. 2213-2217, Nov 2003.

[32] Mathew, S., Anders, M., Krishnamurthy, R.K. and Borkar, S., ”A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core” In IEEE Journal of Solid-State Circuits, Vol. 38, No. 5, pp. 689-695, May 2003.

[33] Grad, J. and Stine, J.E., ”A Multi-Mode Low-Energy Binary Adder,” Fortieth Asilomar Conference on Signals, Systems and Computers, pp. 2065-2068, Oct 2006.

[34] Yan Sun, Dongyu Zheng, Minxuan Zhang and Shaoqing Li, ”High Performance Low-Power Sparse-Tree Binary Adders,” International Conference on Solid-State and Integrated Circuit Technology, pp.1649-1651, Oct 2006.

[35] Tyagi, A., ”A reduced-area scheme for carry-select adders,” IEEE Trans. Comput., Vol. 42, No. 10, pp. 1163-1170, 1993.