Novel Modulo $2^n + 1$ Multipliers *

H. T. Vergos
Computer Engineering and Informatics Dept.,
University of Patras, 26500 Patras, Greece.
vergos@ceid.upatras.gr

C. Efstathiou
Informatics Dept.,TEI of Athens,
12210 Egaleo, Athens, Greece.
cefsta@teiath.gr

Abstract

A new modulo $2^n + 1$ multiplier architecture is proposed for operands in the normal representation. The novel architecture is derived by showing that all required correction factors can be merged into a single constant one and by treating this, partly as a partial product and partly by the final parallel adder. The proposed architecture utilizes a total of $(n + 1)$ partial products, each n bits wide and is built using an inverted end-around-carry, carry-save adder tree and a final parallel adder.

1. Introduction

Residue arithmetic has found great applicability in number theoretic transforms (NTT), that are widely used for convolution / correlation computations [1, 2, 3], cryptographic algorithms [4] and fault-tolerant digital system design. Its main application field is however in the design of specialized digital signal processors that adopt the residue number system (RNS) [5, 6].

In particular, modulo 2^n+1 arithmetic has been the focus of many recent research manuscripts, because this modulus is a part of the well-known three moduli set $\{2^n-1, 2^n, 2^n+1\}$, that has been extensively used in general and special purpose RNS implementations. Efficient modulo $2^n + 1$ component design seems to be the more challenging field for this moduli set, since the modulo $2^n + 1$ components operate on wider operands than the rest two channels and therefore form the execution bottleneck. To attack this problem, Leibowitz [1], introduced the diminished-1 representation. Under this representation each number is represented decreased by 1, while zero operands need to be handled distinctly.

The prime moduli of the form $2^n + 1$ apart from being useful for ordinary RNSs, are vital in the Fermat Number Transform (FNT) and useful in cryptography. More specifically, the Fermat number $2^{16}+1$ was chosen for the modulo multiplier in [2, 3] and for the square–and–multiply modulo exponentiator in an implementation of the International Data Encryption Algorithm [4].

Due to their applicability in the aforementioned fields, numerous architectures have been proposed for modulo $2^n + 1$ components, including adders [7, 8], multi-operand adders and residue generators [9], squarers [10] and multipliers [11, 12, 13, 14, 15, 16, 17, 18, 19].

Although ROM-based structures can be used for computing a modulo $2^n + 1$ product, the exponential growth of the memory required, makes such solutions unsuitable for medium and large values of n. In [11] it was instead proposed that multiplication is carried out using a $(n + 1)$-bit binary multiplier followed by a residue generation circuit. Although suitable even for large n, this solution is not well suited for RNSs that use the $\{2^n - 1, 2^n, 2^n + 1\}$ moduli set, given that the rest channels operate on a $n \times n$ multiplication array. In [12], by observing that groups of the $(n + 1) \times (n + 1)$ partial products array can not be 1 at the same time, the authors managed to reduce the required multiplication array down to $n \times n$. The architecture of [12] however, requires three n-bit parallel adders connected in series followed by a final row of multiplexors. In [13] diminished-1 multipliers with n-bit input operands were considered. Apart from the multiplication array, a circuit that counts the zero partial products. Moreover, handling of zero operands was not attacked.

The first try to apply radix-4 Booth recoding to modulo $2^n + 1$ multiplication appeared in [14]. The partial products used are $(n + 1)$ bits wide. Two correction factors need to be added in an extra carry save adder (CSA) stage for result correction and a diminished-1 final parallel adder with a carry input is also needed. Such an adder is implemented by an additional CSA stage and a diminished-1 parallel adder. The multipliers presented in [15] also use radix-4 Booth recoding but only require n bits for their partial products. These multipliers depart from the diminished-1 discipline, since all operands are in normal form except the

*This research was co-funded by the E.E. (75%) and by the Greek Government (25%), within the framework of the Education and Initial Vocational Training Program : "Archimedes".

Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD'06)
0-7695-2609-8/06 $20.00 © 2006 IEEE

2^n operand which is represented by the all 0s vector. This is done for achieving an efficient design block for use in the international data encryption algorithm (IDEA) block cipher. Although in [15] a scheme is described, for using the multipliers for normal / diminished-1 operands, this implies a dedicated circuit that handles the 2^n value in the case of normal operands or the incorporation of a second parallel modulo adder in that of the diminished-1 operands. Multipliers with Booth recoding in which one operand uses normal representation, while the other the diminished-1 were investigated in [16]. These multipliers are however specific to the cryptographic application targeted.

Diminished-1 multipliers based on a $(n \times n)$ partial products array and a Dadda adder tree were investigated in [17] and were analytically and experimentally shown to outperform those of [13] and [14]. Treatment of zero operands was not however attacked. Modulo $2^n + 1$ multipliers for both diminished-1 and normal operand representation with treatment of zero operands have been presented in [18]. For both cases, Booth recoding is employed to reduce the number of partial products into approximately the half. Both architectures however, require a CSA stage for the addition of a correction factor that is derived by a small dedicated combinational circuit, as well as, a final modulo parallel adder similar to that of [13], that is, with a carry input. Furthermore, in the case of diminished-1 operands, another correction is also introduced. The analytical and experimental results presented in [18] show that the multipliers proposed, outperform the earlier solutions of [14] and [15] if the latter is adopted to diminished-1 or ordinary representation. Comparative results against [17] were not however given. The analytical models for the delay and area presented in [17] and [18], reveal that the architecture of [17] provides smaller multipliers. It also leads to faster designs when $n < 16$ and to designs with the same delay as those described in [18] for $n \geq 16$. This should not be surprising, since it is well-known [15] that although Booth recoding leads to shallower adder tree (about two full adder stages are saved on the critical path when the number of partial products is cut in half), this saving is often compensated or overwhelmed by the recoding logic and the addition of correction factors. Non-Booth encoded modulo $2^n + 1$ multipliers have recently been investigated in [19]. Two architectures have been proposed; one with $n + 3$ partial products and one with $n + 2$ partial products.

In this manuscript, a novel modulo $2^n + 1$ multiplier architecture is presented for input operands in the normal representation. Our architecture is derived following the observations made in [12]. This earlier proposal is however heavily improved by :

- using only one parallel adder instead of three after the reduction of the partial products. This is achieved by analytically deriving a single total correction factor. It is shown that the total correction factor is a constant and therefore, no extra circuit is required to compute it
- splitting the correction factor in two parts. One part is introduced as a partial product, whereas the addition of the second is assigned to the final stage adder. This enables to use a fast parallel-prefix inverted end-around-carry (EAC) parallel adder [7] (equivalently, a diminished-1 modulo $2^n + 1$ adder) as the final stage adder.

The proposed architecture does not use Booth recoding and utilizes a total of $(n + 1)$ n-bit wide partial products. The resulting multipliers obviously outperform the solutions described in [19] in both area and delay, since they use one or two partial products less. The proposed multipliers are compared against those of [17], which according to the discussion above should be considered the currently most efficient ones. The results indicate that the proposed multipliers offer the same or even higher operation speed than those of [17], while being more compact. Considering that the proposed multipliers accept operands in normal form while those of [17] accept operands in diminished-1 representation, it is clear that the proposed multipliers can be used more efficiently since they do not require time- and hardware-consuming input / output translators nor any handling of zero operands.

2. Proposed Architecture

The proposed multiplier architecture is based on merging the correction factors that result from the formation and the reduction of the partial products, into a single correction factor. This is described in detail below.

2.1. Partial Product Formation

Let $A = a_n a_{n-1} \ldots a_1 a_0$ and $B = b_n b_{n-1} \ldots b_1 b_0$ denote two $(n + 1)$-bit numbers in the range $[0, 2^n + 1)$. Obviously, in the normal representation if a_n is 1 then all the rest bits in the representation of A are 0 and the same is true for the rest bits of B, if b_n is 1.

Let $|X|_Y$ denote the modulo Y residue of X. For the multiplication of A with B, it holds :

$$\begin{aligned} R &= \left|A \times B\right|_{2^n+1} = \left|\sum_{i=0}^{n} a_i 2^i \sum_{j=0}^{n} b_j 2^j\right|_{2^n+1} \\ &= \left|\sum_{i=0}^{n}\left(\sum_{j=0}^{n} p_{i,j} 2^{i+j}\right)\right|_{2^n+1} \end{aligned} \quad (1)$$

where the partial product bit $p_{i,j}$ is the logical AND of a_i with b_j. Relation (1) indicates that the partial products (PP)

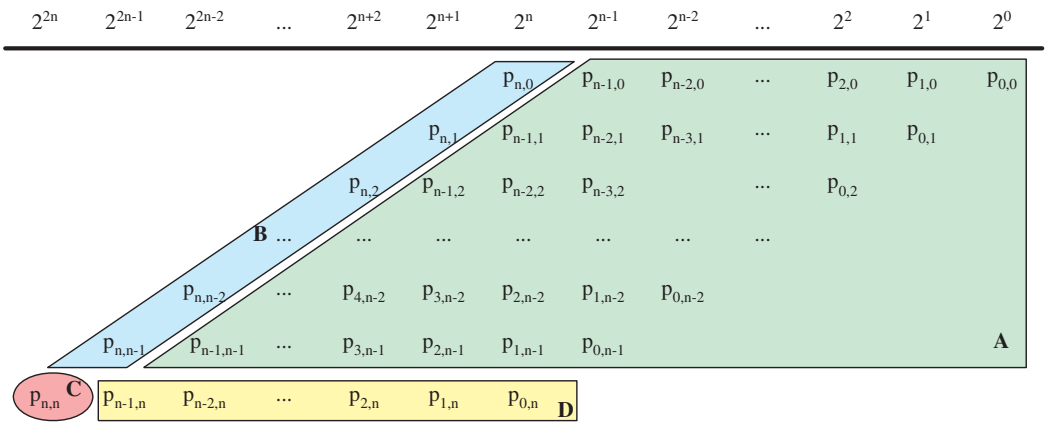

Figure 1. $(n+1) \times (n+1)$ **partial product matrix**

matrix shown in Fig. 1 are required for modulo 2^n+1 multiplication. The partial products matrix can be divided in the four groups A, B, C and D shown in Fig. 1. Note that only terms from one group can be 1 at the same time. Therefore, instead of arithmetically added, terms from different groups can be logically ORed [12].

Combining by logical OR (+, $\oplus$ and $\overline{x}$ are used hereafter to denote logical OR, exclusive-OR and complement of x respectively) the bits of the groups B and D, results into :

$$p_{n,i} + p_{i,n} = (a_n \oplus b_n)(a_i + b_i) = sq_i$$

where $s = a_n \oplus b_n$ and $q_i = a_i + b_i$, with $0 \leq i \leq n-1$. The terms sq_i that overflow to the left of the column with weight 2^{n-1} can then be moved to the column with weight 2^i according to :

$$\begin{aligned} \left|sq_i 2^{n+i}\right|_{2^n+1} &= \left|s(-q_i 2^i)\right|_{2^n+1} = \\ &= \left|s2^i(2^n+1-q_i)\right|_{2^n+1} = \\ &= \left|s\big(2^i(2^n+\overline{q_i})\big)\right|_{2^n+1} = \\ &= \left|s\overline{q_i}2^i + C_i\right|_{2^n+1} \end{aligned}$$

where C_i is a correction factor imposed by each such repositioning, and is equal to $C_i = \left|s2^n 2^i\right|_{2^n+1}$. The correction factor, COR_0, required for repositioning all sq_i terms is given by :

$$COR_0 = \left|\sum_{i=0}^{n-1} C_i\right|_{2^n+1} = \left|s2^n(2^n-1)\right|_{2^n+1} = 2s.$$

COR_0 can be accounted for, by adding s into the column with weight 2^1 of the partial product matrix.

Then, considering the partial products bits of group A, since

$$\begin{aligned} \left|p_{i,j}2^{i+j}\right|_{2^n+1} &= \left|-p_{i,j}2^{|i+j|_n}\right|_{2^n+1} = \\ &= \left|(2^n+1-p_{i,j})2^{|i+j|_n}\right|_{2^n+1} = \\ &= \left|\overline{p_{i,j}}2^{|i+j|_n} + 2^n 2^{|i+j|_n}\right|_{2^n+1} \end{aligned}$$

all $p_{i,j}$ bits with $n \leq (i+j) \leq 2n-1$ can also be inverted and repositioned at the $(i+j-n)$ column. A correction factor of $2^{|i+j|_n}2^n$, should be taken into account for each such complementation and repositioning. For computing the correction factor, COR_1, required for moving all $p_{i,j}$ bits with $n \leq (i+j) \leq 2n-1$, it can be observed that there is only one such bit in the second partial product of Fig. 1, imposing a correction of $2^0 2^n$. The third partial product has two such bits imposing a correction of $(2^0+2^1)2^n = (2^3-1)2^n$ and so on, up to the n-th partial product which will require a correction factor equal to $(2^0+1^1+\cdots+2^{n-2})2^n = (2^{n-1}-1)2^n$. COR_1 is therefore given by summing all these required corrections :

$$\begin{aligned} COR_1 &= 2^n(2(1+2+\cdots+2^{n-2}) - (n-1)) = \\ &= 2^n(2^n-n-1) \end{aligned} \quad (2)$$

Finally, for a completely regular $n \times n$ partial product matrix the term $p_{n,n}$, needs to be repositioned. Since this has a weight of 2^{2n} and $\left|2^{2n}\right|_{2^n+1} = 1$, it can be moved to the rightmost column. Given that $sp_{i,j} = 0$, the terms s and sq_i can be ORed with a term $p_{i,j}$ of the same column.

The above analysis leads to the reduced matrix of partial products (PP_i) presented in Table 1, along with the correction factor indicated by (2) that needs to be taken into account.

Table 1. Reduced Partial Product Matrix

	2^{n-1}	2^{n-2}	$\cdots$	2^2	2^1	2^0
$PP_0 =$	$p_{n-1,0} + s\overline{q_{n-1}}$	$p_{n-2,0} + s\overline{q_{n-2}}$	$\cdots$	$p_{2,0} + s\overline{q_2}$	$p_{1,0} + s\overline{q_1}$	$p_{0,0} + s\overline{q_0} + p_{n,n}$
$PP_1 =$	$p_{n-2,1}$	$p_{n-3,1}$	$\cdots$	$p_{1,1}$	$p_{0,1} + s$	$\overline{p_{n-1,1}}$
$PP_2 =$	$p_{n-3,2}$	$p_{n-4,2}$	$\cdots$	$p_{0,2}$	$\overline{p_{n-1,2}}$	$\overline{p_{n-2,2}}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$PP_{n-2} =$	$p_{1,n-2}$	$p_{0,n-2}$	$\cdots$	$\overline{p_{4,n-2}}$	$\overline{p_{3,n-2}}$	$\overline{p_{2,n-2}}$
$PP_{n-1} =$	$p_{0,n-1}$	$\overline{p_{n-1,n-1}}$	$\cdots$	$\overline{p_{3,n-1}}$	$\overline{p_{2,n-1}}$	$\overline{p_{1,n-1}}$

2.2. Partial product reduction

The n partial products of Table 1 and the total correction factor, that is, $n+1$ partial products in total, must be added modulo 2^n+1, until two final summands are produced. This can be performed by using either a CSA adder array or a CSA adder tree (Dadda tree [20]). It is well-known, that in integer multipliers a CSA tree results to irregular architectures. However, in our case, the resulting array is completely regular, that is, well-suited for VLSI implementations. This is because, the same number of bits exists in every column and the carry outputs at the most significant bit position of each stage, can be used inverted, as carry inputs of the subsequent stage.

Suppose that the carry output of the n-th column of stage i is denoted by c_i. This signal has a weight of 2^n. Since :

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

the carries out of the most significant bit position can be complemented and added to the least significant bit position of the next stage, forming an inverted EAC CSA tree. A correction factor of 2^n must be taken into account for each such carry recirculation. During the addition of our $n+1$ partial products, $n-1$ carries of weight 2^n will be generated and therefore the correction, COR_2, that would be required for the inverted EACs is :

$$COR_2 = |2^n(n-1)|_{2^n+1}. \quad (3)$$

The total correction required is consequently given by the addition of the factors derived in (2) and (3) :

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

Therefore, equation (1) takes the form :

$$R = |A \times B|_{2^n+1} = \left| \sum_{i=0}^{n-1} PP_i + 3 \right|_{2^n+1}. \quad (4)$$

where the n partial products have been presented in Table 1.

2.3. Final stage addition

A straightforward implementation that one can easily derive from (4), is to use COR as an extra partial product, along with a fast modulo 2^n+1 adder (for example [8]) which accepts the two summands produced by the reduction scheme (array / tree) and produces the product. An alternative, more efficient solution is however proposed in the following, based on the use of an inverted EAC parallel adder (equivalently, a diminished-1 modulo 2^n+1 adder). If the architecture proposed in [7] is followed, this adder can provide its result faster than the fastest modulo 2^n+1 adder available [8], with smaller area requirements. Since however such an adder can provide only n bits of the result, the remaining bit must be derived in a separate manner.

Given that, if two n-bit operands, suppose S and C, with normal representation are used as inputs to a an inverted EAC parallel adder, its output will be equal to $|S+C+1|_{2^n+1}$, it becomes obvious that for using an inverted EAC parallel adder, part of the correction factor should be assigned to the final adder, while the rest is treated as a partial product. This leads to rewrite (4) as :

$$\begin{aligned} R = |A \times B|_{2^n+1} &= \left| \sum_{i=0}^{n-1} PP_i + 3 \right|_{2^n+1} = \\ &= \left| \left| \sum_{i=0}^{n-1} PP_i + 2 \right|_{2^n+1} + 1 \right|_{2^n+1} \quad (5) \end{aligned}$$

Let $S = s_{n-1}s_{n-2}\ldots s_0$ and $C = c_{n-2}c_{n-3}\ldots c_0\overline{c_{n-1}}$ denote the sum and carry n-bit vectors that are produced by the multi-operand addition $\left|\sum_{i=0}^{n-1} PP_i + 2\right|_{2^n+1}$, that is, $S + C = \left|\sum_{i=0}^{n-1} PP_i + 2\right|_{2^n+1}$. Substituting this in (5), results into :

$$R = |A \times B|_{2^n+1} = |S+C+1|_{2^n+1}. \quad (6)$$

Since the most significant bit of the multiplication, is 1 only when $R = 2^n$, from (6) it is derived that $R = 2^n <=> |S+C+1|_{2^n+1} = 2^n$. Taking into account that S and C are n-bit vectors we get that $R = 2^n <=> S+C+1 = 2^n$ or equivalently that $S + C = 2^n - 1$. That is, the most

significant bit of the multiplication is 1 only when S and C are complementary vectors. As explained at the end of this subsection, this observation enables to compute the most significant bit distinctly from the rest. In the following we focus on the n least significant bits of R.

Let R_n denote the n-bit vector of the least significant bits of R. Since :

$$\begin{aligned} R_n &= \big||A \times B|_{2^n+1}\big|_{2^n} = \big||S+C+1|_{2^n+1}\big|_{2^n} \\ &= \begin{cases} |S+C-2^n|_{2^n}, & \text{if } S+C \geq 2^n \\ |S+C+1|_{2^n}, & \text{otherwise} \end{cases} \\ &= \begin{cases} |S+C|_{2^n}, & \text{if } S+C \geq 2^n \\ \big||S+C|_{2^n}+1\big|_{2^n}, & \text{otherwise} \end{cases} \end{aligned}$$

the n least significant bits of the product can be handled by an n-bit adder that increases the binary sum of its inputs by one when the carry output is 0 and leaves it unchanged in the case of a carry output. This is exactly the function performed by an inverted EAC parallel adder. Therefore, if a total correction factor of 2 is used as an extra partial product, an inverted EAC final parallel adder will accept S and C at its inputs and will provide R_n.

Very fast inverted EAC adders based on parallel prefix carry computation units have appeared in [7, 15]. For integer adders, a parallel prefix carry computation unit is derived based on the following. Let $A = a_{n-1}a_{n-2}\ldots a_1a_0$ and $B = b_{n-1}b_{n-2}\ldots b_1b_0$ denote the two n-bit addition operands and the let the terms $g_i = a_ib_i$, $p_i = a_i + b_i$ and $h_i = a_i \oplus b_i$ denote the carry generate, the carry propagate and the half–sum terms at bit position i respectively. By defining the $\circ$ operator as an operator that associates generate and propagate pairs and produces a new pair according to the equation :

$$(g_x, p_x) \circ (g_y, p_y) = (g_x + p_x g_x, p_x p_y)$$

the computation of a carry c_i of the integer addition of A and B is equivalent to the problem of computing G_i, where :

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

Once the carries have been computed the sum bits, s_i, are computed by $s_i = h_i \oplus c_{i-1}$.

For attaining an inverted EAC adder, simple solutions, such as the connection of the carry output of an integer adder back to the carry input via an inverter, are not well-suited, since they suffer from oscillations. In [15], it was proposed that the output carry is driven back via an inverter to a late carry increment stage composed of nodes implementing a prefix operator. Therefore, no oscillations occur and the derived inverted EAC adders feature an operating speed close to the corresponding integer adders.

The need for an extra prefix stage that handles the re-entering carry has been cancelled in the parallel-prefix inverted EAC adders proposed in [7]. This was achieved by performing carry re-circulation at each existing prefix level. As a result parallel-prefix adder architectures with $\log_2 n$ have been derived, that is, inverted EAC adders that can achieve the same operating speed as the corresponding integer adders. For the sake of completeness, some of the theory developed in [7] is revisited in the following.

The inverted EAC adder carry, suppose c_i^*, is equal to G_i^*, where :

$$(G_i^*, P_i^*) =$$

$$= \begin{cases} \overline{(G_{n-1}, P_{n-1})}, & \text{if } i = -1 \\ (G_i, P_i) \circ \overline{(G_{n-1,i+1}, P_{n-1,i+1})}, & \text{otherwise,} \end{cases} \quad (7)$$

and :

- $\overline{(G, P)} = (\overline{G}, P)$
- $G_{a,b}$ and $P_{a,b}$, with $a > b$, are respectively the group generate and propagate signals for the group a, $a-1$, $a-2$, ..., $b+1$, b, computed by : $(G_{a,b}, P_{a,b}) = (g_a, p_a) \circ (g_{a-1}, p_{a-1}) \circ \cdots \circ (g_b, p_b)$.

In the cases that the equations indicated by (7) require more than $\log_2 n$ prefix levels for their implementation, they are transformed them into equivalent ones by introducing t_i, $t_i = a_i + b_i$, and taking into account that if $(G^x, P^x) = (g, p) \circ \overline{(G, P)}$ and $(G^y, P^y) = \overline{((\overline{t}, \overline{g}) \circ (G, P))}$ then $G^x = G^y$ [7]. This enables to equivalently compute a carry whose equation is given by a prefix equation of the form $(g, p) \circ \overline{(G, P)}$ as $\overline{((\overline{t}, \overline{g}) \circ (G, P))}$. For area-time efficient designs this transformation should be applied j times recursively to the equations of the form $(g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \cdots \circ (g_1, p_1) \circ \overline{(G_{n,i+1}, P_{n,i+1})}$ produced by (7), until:

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

Since the inputs of the final adder are S and C and the most significant bit of the result should be 1, only in the case that S and C are complementary vectors, it is obvious that this is equal to the group propagate signal out of the n bits $P_{n-1} = p_{n-1}p_{n-2}\cdots p_0$, of the final inverted EAC adder.

2.4. An example of the proposed architecture

Table 2 lists the required partial products in a modulo 17 multiplier that follows the proposed architecture. The last partial product represents the total correction factor when an inverted EAC adder is used as the final adder.

Table 2. Partial Product Matrix in Modulo 17 Multiplication

	2^3	2^2	2^1	2^0
$PP_0 =$	$p_{3,0} + s\overline{q_3}$	$p_{2,0} + s\overline{q_2}$	$p_{1,0} + s\overline{q_1}$	$p_{0,0} + s\overline{q_0} + p_{4,4}$
$PP_1 =$	$p_{2,1}$	$p_{1,1}$	$p_{0,1} + s$	$\overline{p_{3,1}}$
$PP_2 =$	$p_{1,2}$	$p_{0,2}$	$\overline{p_{3,2}}$	$\overline{p_{2,2}}$
$PP_3 =$	$p_{0,3}$	$\overline{p_{3,3}}$	$\overline{p_{2,3}}$	$\overline{p_{1,3}}$
$PP_4 =$	0	0	1	0

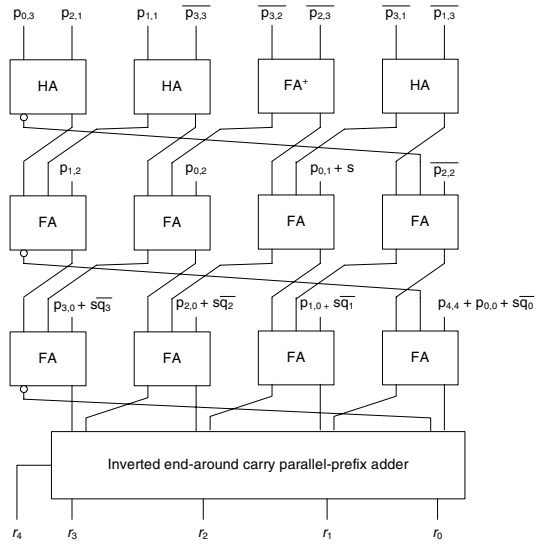

Figure 2. Proposed architecture of a modulo 17 multiplier.

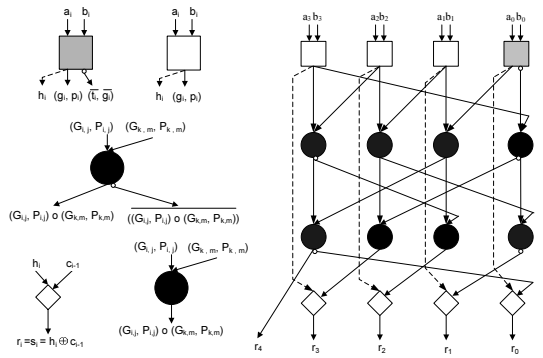

Figure 3. Final adder of the proposed modulo 17 multiplier.

Figure 2 presents a block diagram of the proposed modulo 17 multiplier. The simple gates required for the formation of the partial product bits are not shown. The blocks used are half-adders (HA), full-adders (FA), simplified FA blocks (FA$^+$), that is, FAs with one of their inputs set at 1 and the final adder. The output carries at the most significant bit of each stage are complemented and driven to the least significant bit of the subsequent stage. The two final derived summands are added in the final inverted EAC adder.

It should be noted that the partial product bits of equal weight should not be driven randomly in the FAs of the corresponding column. For achieving the least delay, the partial product bits derived earlier should be driven to the FAs at the top of the CSA tree, whereas late arriving signals to FAs of subsequent tree levels. For example, the FAs of the rightmost column in Figure 2, perform the addition of $0, \overline{p_{3,1}}, \overline{p_{1,3}}, \overline{p_{2,2}}$ and $p_{4,4} + p_{1,0} + s\overline{q_0}$ along with the inverted carries that overflow at the leftmost column. The addition of 0 can not be avoided since doing so alters the number of inverted EACs in the CSA tree and invalidates all the previous analysis. However, the FAs accepting the bits from the constant partial product can be simplified to HAs or FA$^+$. Since it is expected that the signals $\overline{p_{3,1}}, \overline{p_{1,3}}$ and $\overline{p_{2,2}}$ would be the ones computed earlier, these along with the 0 operand are those should be examined as candidates for the first addition stage.

The final adder required in this case is shown in Fig. 3. The parallel-prefix carry computation unit of the adder computes the modulo $2^n + 1$ carries according to the following expressions :

$$
\begin{aligned}
c^*_{-1} &\longleftrightarrow \overline{(g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1) \circ (g_0, p_0)} \\
c^*_0 &\longleftrightarrow \overline{(\overline{t_0}, \overline{g_0}) \circ (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1)} \\
c^*_1 &\longleftrightarrow (g_1, p_1) \circ (g_0, p_0) \circ \overline{((g_3, p_3) \circ (g_2, p_2))} \\
c^*_2 &\longleftrightarrow (g_2, p_2) \circ (g_1, p_1) \circ \overline{((\overline{t_0}, \overline{g_0}) \circ (g_3, p_3))}
\end{aligned}
$$

From Fig. 2 it is evident that the proposed multipliers have a very regular structure that is well suited for VLSI implementation.

3. Area - delay analysis and comparisons

In this section, the area and time complexities of the proposed multipliers are analyzed. The proposed multipliers are compared qualitatively and quantitatively against the multipliers proposed in [17]. For our qualitative comparisons, we use the unit-gate model proposed in [21]. This model considers that all 2-input monotonic gates count as one equivalent for both area and delay, whereas a 2-input XOR/XNOR gate counts as two equivalents.

The area requirements of the proposed multipliers consist of the gates forming the partial product bits, the CSA tree and the final adder. Since $n^2 + 1$ AND (NAND) gates are needed for the formation of the $p_{i,j}$ ($\overline{p_{i,j}}$) bits, n NOR gates for the $\overline{q_i}$ bits, 1 XOR gate for s, n AND gates for

Table 3. Area Comparisons

n	4	8	12	16	20	24	28	32
$A_{proposed}$	145	571	1,289	2,227	3,511	4,979	6,703	8,683
A[17]	170	628	1,356	2,348	3,603	5,120	6,896	8,932
Savings (%)	14.7	9.1	4.9	5.2	2.6	2.8	2.8	2.8

the $s\overline{q_i}$ bits and finally $n+2$ OR gates are needed for the $p_{i,j}+s\overline{q_i}$, $p_{i,j}+s$ and $p_{0,0}+p_{n,n}+s\overline{q_0}$ bits, the area required for the partial product bits formation is modeled as :

$$A_{partial} = n^2+1+n+2+n+n+2 = n^2+3n+5 \quad (8)$$

gate equivalents. The CSA tree that performs the reduction of the partial products in two final summands is composed of $(n-1)$ rows of n FAs each. However, since one of these rows accepts the constant correction factor its n FAs can be simplified to $n-1$ HAs and one FA$^+$. Considering that an FA, an HA and an FA$^+$ have an area of 7, 3 and 3 gate equivalents respectively, we get that the area required for the partial products reduction is :

$$A_{CSA} = 7n(n-2)+3(n-1)+3 = 7n^2-11n \quad (9)$$

gate equivalents. The area of the last stage adder was computed in [7] as

$$A_{adder} = \frac{9}{2}n\log_2 n + \frac{1}{2}n + 6 \quad (10)$$

Summing the requirements of (8), (9) and (10) we get that the proposed multipliers have an area of :

$$A_{proposed} = 8n^2 + \frac{9}{2}n\log_2 n - \frac{15}{2}n + 11$$

whereas the area of the multipliers proposed in [17] was :

$$A[17] = 8n^2 + \frac{9}{2}n\log_2 n + \frac{1}{2}n + 4$$

equivalent gates. We therefore conclude that the proposed architecture leads to more compact designs. Table 3, lists for various operand sizes the savings in area offered by the proposed multipliers.

The delay of the proposed multipliers also consists of three parts, namely, the delay of the partial product formation, the delay of the CSA tree and the delay of the parallel inverted EAC adder. Since, the $p_{i,j}$ and $(\overline{p_{i,j}})$ terms are computed in 1 time unit, the term $p_{i,j}+s$ in 3 time units and the terms $p_{i,j}+s\overline{q_i}$ and $p_{0,0}+p_{n,n}+s\overline{q_0}$ in 4 time units, the upper bound of the delay for forming the partial product matrix is 4 time units. However, as explained previously, in some cases we can parallelize some of this delay with that of the first stage of the CSA tree by driving the late arriving partial product bits to the FAs of subsequent stages. Taking into account that the delay of a FA is 4 time units and that when $n+1$ is a number of the Dadda sequence (6, 9, 13, 19, 28, 42, 63, ...), for an optimal depth CSA tree, all partial product bits are required in the first stage of the tree, we can model the contribution of the partial product formation as :

Table 4. Delay Comparisons

n	4	8	12	16	20	24	28	32
$T_{proposed}$	19	29	35	36	42	42	46	46
T[17]	24	28	34	36	42	42	46	46

$$T_{partial} = \begin{cases} 4, & \text{if } n+1 \text{ is a Dadda number,} \\ 1, & \text{otherwise} \end{cases} \quad (11)$$

time units. The delay of a CSA tree designed according to Dadda [20] can be modelled as:

$$T_{CSA} = 4D(n+1) \quad (12)$$

where $D(k)$ denotes the depth in FAs of a k-operand CSA tree, available in table 5.1 of [22]. In the special case of $n=4$, since the first stage of the tree can be constructed using only HA and FA$^+$ blocks, we have that $T_{CSA}=10$. The delay of the parallel-prefix inverted EAC adder is [7] :

$$T_{adder} = 2\log_2 n + 3. \quad (13)$$

Summing the delays of (11), (12) and (13), we conclude that the delay of the proposed multipliers can be modeled as :

$$T_{proposed} = \begin{cases} 19, & \text{if } n=4, \\ 4D(n+1)+2\log_2 n+7, & \text{if } n+1 \text{ is a Dadda number,} \\ 4D(n+1)+2\log_2 n+4, & \text{otherwise.} \end{cases}$$

time units. The multipliers of [17] exhibit a delay of

$$T[17] = \begin{cases} 4D(n+3)+2\log_2 n+2, & \text{if } n+1 \text{ or } n+2 \text{ is a Dadda number,} \\ 4D(n+3)+2\log_2 n+4, & \text{otherwise.} \end{cases}$$

Table 4, lists for various operand sizes the delays offered by the proposed architecture and the architecture of [17]. In most cases the two architectures offer similar execution delay. However, the proposed architecture operates

Table 5. Area and delay results from cell-based implementations

	Multipliers of [17]		Proposed Multipliers	
n	*Area*	*Delay)*	*Area*	*Delay*
4	3724	1.34	3031	1.12
8	9685	1.98	8721	1.98
16	36541	2.65	34287	2.62
32	118552	3.63	113912	3.60

on operands with normal representation and does not therefore require any input / output converters as the architecture of [17] does, which increase further its delay.

Since the unit-gate model does not take into account the interconnect complexity, as well as, fan-in and fan-out requirements of the compared designs, a cell-based design approach was used to verify the qualitative comparisons. Each multiplier was described in structural HDL using cells from a 0.18μm CMOS standard cell library. After an initial mapping, the synthesis tool performed iterative steps of delay optimization on each design until no faster netlist could be reached. Recursive area recovery steps were then applied. The netlists and the associated constraints were then passed to a standard cell place and route tool. All design constraints such as output load, max fan-out and floorplan initialization were kept constant during each comparison. After the netlists were annotated with the back-end information, static timing analysis was performed. The attained results are listed in Table 5. The reported area results include both cell and interconnect area and are given in μm^2. The delay results are in nanoseconds. On the average of the examined cases, the proposed architecture leads to 9.7% more compact and in parallel 4.6% faster multipliers.

4. Conclusions

A novel architecture for a modulo $2^n + 1$ multiplier was proposed in this paper. The proposed architecture improves heavily the one proposed in [12] by combining all corrections into a single one, thereby decreasing the required parallel additions from 3 to 1. Even higher speed is attained by treating part of the required correction as a partial product while the rest is handled by the last stage adder.

Our comparisons against the most efficient known multipliers indicate that the proposed multipliers offer the same or even higher speed while being more compact. Since the proposed multipliers accept operands in normal form, they do not require time- and hardware-consuming input / output translators and special handling of zero operands as the earlier solution which accepts diminished-1 operands. Therefore, the proposed multipliers can be used more efficiently.

References

[1] L. M. Leibowitz. A Simplified Binary Arithmetic for the Fermat Number Transform. *IEEE Trans. Acoust., Speech, Signal Processing*, 24:356–359, 1976.

[2] M. Benaissa et al. Diminished-1 Multiplier for a Fast Convolver and Correlator using the Fermat Number Transform. *IEE Proceedings G*, 135:187–193, 1988.

[3] S. Sunder et al. Area-efficient Diminished-1 Multiplier for Fermat Number-theoretic Transform. *IEE Proceedings G*, 140:211–215, 1993.

[4] R. Zimmermann et al. A 177 Mb/s VLSI Implementation of the International Data Encryption Algorithm. *IEEE J. Solid-State Circuits*, 29(3):303–307, 1994.

[5] E. D. Claudio et al. Fast Combinatorial RNS Processors for DSP Applications. *IEEE Trans. Comput.*, 44:624–633, 1995.

[6] J. Ramirez et al. High Performance, Reduced Complexity Programmable RNS–FPL Merged FIR Filters. *Electronics Letters*, 38(4):199–200, 2002.

[7] H. T. Vergos et al. Diminished-One Modulo $2^n + 1$ Adder Design. *IEEE Trans. Comput.*, 51:1389–1399, 2002.

[8] C. Efstathiou et al. Fast Parallel-Prefix Modulo $2^n + 1$ Adders. *IEEE Trans. Comput.*, 53:1211–1216, 2004.

[9] S. J. Piestrak. Design of Residue Generators and Multioperand Modular Adders using Carry-Save Adders. *IEEE Trans. Comput.*, 43:68–77, 1994.

[10] H. T. Vergos and C. Efstathiou. Diminished-1 Modulo 2^n+1 Squarer Design. *IEE Proceedings - Computers and Digital Techniques*, 152:561–566, 2005.

[11] A. A. Hiasat. A Memoryless $mod(2^n \pm 1)$ Residue Multiplier. *Electronics Letters*, 28(3):314–315, 1992.

[12] A. Wrzyszcz and D. Milford. A New Modulo $2^a + 1$ Multiplier. In *Proc. of the International Conference on Computer Design (ICCD'93)*, pages 614–617, 1993.

[13] Z. Wang et al. An Efficient Tree Architecture for Modulo $2^n + 1$ Multiplication. *Journal of VLSI Signal Processing*, 14:241–248, 1996.

[14] Y. Ma. A Simplified Architecture for Modulo $(2^n + 1)$ Multiplication. *IEEE Trans. Comput.*, 47(3):333–337, 1998.

[15] R. Zimmerman. Efficient VLSI Implementation of Modulo $(2^n \pm 1)$ Addition and Multiplication. In *Proc. of the* 14^{th} *IEEE Symposium on Computer Arithmetic*, pages 158–167, April 1999.

[16] A. Curiger. *VLSI Architectures for Computations in Finite Rings and Fields*. PhD thesis, Swiss Federal Institute of Technology, 1993.

[17] C. Efstathiou et al. Efficient Diminished-1 Modulo $2^n + 1$ Multipliers. *IEEE Trans. Comput.*, 54:491–496, 2005.

[18] L. Sousa and R. Chaves. A Universal Architecture for Designing Efficient Modulo $2^n + 1$ Multipliers. *IEEE Trans. Circuits Syst. I*, 52:1166–1178, 2005.

[19] R. Chaves and L. Sousa. Faster Modulo $2^n + 1$ Multipliers without Booth Recoding. In *Proc. of the XX Conference on Design of Circuits and Integrated Systems (DCIS '05)*, 2005.

[20] L. Dadda. On Parallel Digital Multipliers. *Alta Frequenza*, 45:574–580, 1976.

[21] A. Tyagi. A Reduced-Area Scheme for Carry-Select Adders. *IEEE Trans. Comput.*, 42(10):1163–1170, 1993.

[22] I. Koren. *Computer Arithmetic Algorithms,* 2^{nd} *edition*. A. K. Peters, Natick, 2002.