Efficient Diminished-1 Modulo $2^n + 1$ Multipliers

Costas Efstathiou,
Haridimos T. Vergos, *Member*, *IEEE*,
Giorgos Dimitrakopoulos, and
Dimitris Nikolos, *Member*, *IEEE*

Abstract—In this work, we propose a new algorithm for designing diminished-1 modulo $2^n + 1$ multipliers. The implementation of the proposed algorithm requires $n + 3$ partial products that are reduced by a tree architecture into two summands, which are finally added by a diminished-1 modulo $2^n + 1$ adder. The proposed multipliers, compared to existing implementations, offer enhanced operation speed and their regular structure allows efficient VLSI implementations.

Index Terms—Modulo $2^n + 1$ multipliers, computer arithmetic, residue number system, Fermat number transform, VLSI design.

1 Introduction

Arithmetic modulo $2^n + 1$ has been used in several applications, which include specialized digital signal processors based on Residue Number System (RNS) arithmetic [1], [2], [3], [4], Fermat Number Transform (FNT) for eliminating the roundoff errors in convolution computations [5], [6], [7], [8], and cryptographic algorithms [9]. For the implementation of these applications, several designs for modulo $2^n + 1$ arithmetic blocks have been proposed. Efficient modulo $2^n + 1$ adders have been presented in [10], [11], [12], multioperand adders and residue generators in [13], and multipliers in [14], [15], [16], [17], [18]. The prime moduli of the form $2^n + 1$, apart from being useful for ordinary RNSs, are vital in FNT and useful in cryptography. The Fermat number $2^{16} + 1$, by being the only Fermat number of practical interest, was chosen for the implementation of the International Data Encryption Algorithm (IDEA) [9].

Since a number in the range of $[0, 2^n]$ requires $n + 1$ bits for its representation, the weighted representation of an operand modulo $2^n + 1$ is a problem in an RNS that uses the three moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, given that the other two channels operate on n-bit quantities. To overcome this problem and since, in the case of a zero operand, the result can be derived straightforwardly, Leibowitz [5] introduced the diminished-1 representation. Under this representation, each number is represented decremented by 1 modulo $2^n + 1$ and all arithmetic operations are inhibited for a zero operand. Zero is represented using a separate zero indication bit. This representation has the advantage that the numbers are represented by n bits and simplifies the basic operations of addition, multiplication, and scaling modulo $2^n + 1$. Recently, the benefits of diminished-1 arithmetic have been utilized for the design of low-power convolution architectures [19] and for high speed implementation of the IDEA cryptographic algorithm [20].

- *C. Efstathiou is with the Department of Informatics, TEI of Athens, Ag. Spyridonos St., 12210 Egaleo, Athens, Greece. E-mail: cefsta@teiath.gr.*
- *H.T. Vergos, G. Dimitrakopoulos, and D. Nikolos are with the Technology and Computer Architecture Lab, Computer Engineering and Informatics Department, University of Patras, 26500 Patras, Greece. E-mail: {vergos, dimitrak}@ceid.upatras.gr, nikolosd@cti.gr.*

Manuscript received 31 Oct. 2003; revised 18 June 2004; accepted 22 Nov. 2004; published online 15 Feb. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0198-1003.

We can distinguish the multipliers modulo $2^n + 1$ in the following categories, depending on the type of operands that they accept:

- Both operands use standard representation [14], [15].
- One input uses a standard representation, while the other utilizes a diminished-1 representation [18].
- Both inputs use diminished-1 representation [16], [17].

It is important to note that the multipliers presented in [10] also use n bits for their representation, but do not follow the diminished-1 discipline. This representation is specific for the IDEA implementation and imposes all operands to be in weighted form, except the operand 2^n, which is represented as an all zeros operand.

In this paper, we present a new algorithm for designing tree multipliers for the third of the above categories, that is, modulo $2^n + 1$ multipliers whose both inputs are in diminished-1 representation. We show that the proposed multipliers are more efficient than the multipliers presented in [14], [15], [16], [17], [18]. The new design method is presented in Section 2. An area and delay analysis is given in Section 3 and compared against the previous solutions. Experimental results based on static CMOS implementations are also presented in Section 3. Our conclusions are drawn in the last section.

2 The Proposed Multipliers

In this section, a new architecture for modulo $2^n + 1$ multiplication for diminished-1 operands is introduced. At first, the derivation of the partial products is explained. Then, the reduction of the partial products in two summands is examined.

Let A, B be two $(n+1)$-bit numbers with $0 \leq A, B < 2^n + 1$ and suppose that $A_{-1} = a_{n-1}a_{n-2}\ldots a_0$, $B_{-1} = b_{n-1}b_{n-2}\ldots b_0$ denote their diminished-1 representations such that

$$A_{-1} = \left|A - 1\right|_{2^n+1} \qquad B_{-1} = \left|B - 1\right|_{2^n+1} \tag{1}$$

and $A_{-1}, B_{-1} \neq 0$. Assume that Q denotes the product of A and B modulo $2^n + 1$, that is, $Q = |A \times B|_{2^n+1}$, where $|x|_m$ denotes the residue of x modulo m. Then, according to [16] and [10], for the diminished-1 representation of Q, we have that

$$\begin{aligned}\left|Q_{-1}\right|_{2^n+1} = \left|Q - 1\right|_{2^n+1} &= \left|A \times B - 1\right|_{2^n+1} \\ &= \left|(A_{-1}+1)\times(B_{-1}+1) - 1\right|_{2^n+1} \\ &= \left|A_{-1}\times B_{-1} + A_{-1} + B_{-1}\right|_{2^n+1} \\ &= \left|\left|A_{-1}\times B_{-1}\right|_{2^n+1} + A_{-1} + B_{-1}\right|_{2^n+1}.\end{aligned} \tag{2}$$

The term $|A_{-1} \times B_{-1}|_{2^n+1}$ of (2) can be expressed as

$$\begin{aligned}\left|A_{-1}\times B_{-1}\right|_{2^n+1} &= \left|\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j 2^{i+j}\right|_{2^n+1} \\ &= \left|\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j \left|2^{i+j}\right|_{2^n+1}\right|_{2^n+1}.\end{aligned} \tag{3}$$

Taking into account that $i + j \leq 2n - 2$, (3) can be written as

$$\left|A_{-1}\times B_{-1}\right|_{2^n+1} = \left|\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j (-1)^s 2^{|i+j|_n}\right|_{2^n+1}, \tag{4}$$

where

$$s = \begin{cases} 0, & \text{if } i+j < n \\ 1, & \text{if } i+j \geq n. \end{cases} \quad (5)$$

For the two cases of (5), relation (4) can be expressed as

$$\left|A_{-1} \times B_{-1}\right|_{2^n+1} = \left| \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} a_i b_j \, 2^{|i+j|_n} + \sum_{i=1}^{n-1} \sum_{j=n-i}^{n-1} (-a_i b_j) \, 2^{|i+j|_n} \right|_{2^n+1}. \quad (6)$$

For $z \in \{0, 1\}$, it holds that

$$\left| -z \right|_{2^n+1} = \left|2^n + 1 - z\right|_{2^n+1} = \left|2^n + \overline{z}\right|_{2^n+1}, \quad (7)$$

where $\overline{z}$ denotes the complement of bit z. Then, according to (7), (6) can be rewritten as

$$\left|A_{-1} \times B_{-1}\right|_{2^n+1} = \left| \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} a_i b_j \, 2^{|i+j|_n} + \sum_{i=1}^{n-1} \sum_{j=n-i}^{n-1} (2^n + \overline{a_i b_j}) \, 2^{|i+j|_n} \right|_{2^n+1}. \quad (8)$$

Relation (8) indicates that one way to form the partial products is to complement each bit $a_i b_j$ with $i + j \geq n$ and place it at bit position $|i + j|_n$, provided that a correction equal to $|2^n 2^{|i+j|_n}|_{2^n+1}$ is taken into account for each complementation. Therefore, (8) can be reformulated as

$$\left|A_{-1} \times B_{-1}\right|_{2^n+1} = \left| \sum_{i=0}^{n-1} (PP_i + C_i) \right|_{2^n+1}, \quad (9)$$

where PP_i denotes the ith partial product

$$PP_i = \begin{cases} \sum_{j=0}^{n-1} a_0 b_j \, 2^j, & \text{if } i = 0 \\ \sum_{j=0}^{n-1-i} a_i b_j \, 2^{|i+j|_n} + \sum_{j=n-i}^{n-1} \overline{a_i b_j} \, 2^{|i+j|_n}, & \text{if } i \neq 0 \end{cases} \quad (10)$$

and C_i is the corresponding correction factor. It should be noted that PP_0 does not contain any complemented bits and, thus, $C_0 = 0$. On the other hand, for $i \neq 0$, the value of C_i depends on the number of complemented bits $\overline{a_i b_j}$ and is given by

$$C_i = \sum_{j=n-i}^{n-1} \left|2^n \, 2^{|i+j|_n}\right|_{2^n+1} = 2^n \, (2^i - 1). \quad (11)$$

According to (10) and (11), the following partial products and correction factors are derived:

$$\begin{array}{lllllll}
PP_0 = & a_0 b_{n-1} & a_0 b_{n-2} & \ldots & a_0 b_1 & a_0 b_0, & C_0 = \; 0 \\
PP_1 = & a_1 b_{n-2} & a_1 b_{n-3} & \ldots & a_1 b_0 & \overline{a_1 b_{n-1}}, & C_1 = \; 2^n \, (2^1 - 1) \\
PP_2 = & a_2 b_{n-3} & a_2 b_{n-4} & \ldots & \overline{a_2 b_{n-1}} & \overline{a_2 b_{n-2}}, & C_2 = \; 2^n \, (2^2 - 1) \\
\ldots & & & & & & \\
PP_{n-2} = & a_{n-2} b_1 & a_{n-2} b_0 & \ldots & \overline{a_{n-2} b_3} & \overline{a_{n-2} b_2}, & C_{n-2} = \; 2^n \, (2^{n-2} - 1) \\
PP_{n-1} = & a_{n-1} b_0 & \overline{a_{n-1} b_{n-1}} & \ldots & \overline{a_{n-1} b_2} & \overline{a_{n-1} b_1}, & C_{n-1} = \; 2^n \, (2^{n-1} - 1).
\end{array}$$

The total correction, C_P, required for the formation of the above n partial products is equal to

$$C_P = \sum_{i=0}^{n-1} C_i = C_0 + \sum_{i=1}^{n-1} 2^n \, (2^i - 1) = 2^n (2^n - 1 - n). \quad (12)$$

In the following, we consider the reduction of the partial products into two summands. This can be performed in a variety of ways. In this paper, an FA-based Dadda tree architecture is followed [21]. Although the use of a tree architecture in integer multipliers results in irregular architectures, in our case, the resulting FA array is completely regular and, therefore, well-suited for VLSI implementations. This is due to the fact that the same number of bits participate in every bit position since the carry output of the most-significant bit position is fed back as a carry input to the least-significant bit position of the next stage. Let c_n denote a carry output at the most significant bit position which has a weight of 2^n. Since

$$\left|c_n 2^n\right|_{2^n+1} = \left| - c_n\right|_{2^n+1} = \left|2^n + \overline{c}_n\right|_{2^n+1}, \quad (13)$$

then c_n can be complemented and added at the least significant bit position of the next stage, provided that a correction of 2^n is taken into account. Since an FA row reduces the number of partial products by one, $n + 1$ FA rows are required in order to derive the two final summands from $n + 3$ partial products. The FAs at the most significant bit position will then produce $n + 1$ carries of weight 2^n. Therefore, the correction, C_R, required during the addition of $n + 3$ partial products is

$$C_R = (n + 1) \, 2^n. \quad (14)$$

Merging both correction factors of (12) and (14) results in a single factor C, which, in modulo $2^n + 1$ arithmetic, is equal to

$$\left|C\right|_{2^n+1} = \left|C_P + C_R\right|_{2^n+1} = \left|2^n(2^n - n - 1) + (n + 1)2^n\right|_{2^n+1} = 1. \quad (15)$$

Since C is treated in the proposed architecture as an extra partial product, we have to use its diminished-1 representation in our reduction scheme, i.e., $C_{-1} = |C - 1|_{2^n+1}$, which is equal to the all 0s n-bit vector. This vector, along with the n PP_is of (9) and the A_{-1}, B_{-1} of (2) forms the $n + 3$ partial products of the proposed architecture. Although $C_{-1} = 0$, it cannot be ignored during the reduction of the partial products since, in this case, less than $n + 1$ carries of weight 2^n will be produced. The above analysis indicates that

$$\left|Q_{-1}\right|_{2^n+1} = \left| \sum_{i=0}^{n-1} PP_i + A_{-1} + B_{-1} + C_{-1} \right|_{2^n+1}. \quad (16)$$

An implementation of the proposed architecture is composed of AND or NAND gates that form a bit of each partial product, a Dadda tree that reduces the $n + 3$ partial products into two summands, and a modulo $2^n + 1$ adder for diminished-1 operands [12] that accepts these two summands and produces the required product.

A diminished-1 modulo $2^n + 1$ parallel adder is effectively an inverted end-around-carry adder. Since a direct connection of the carry output to the carry input via an inverter leads to an oscillating circuit, dedicated architectures have been proposed that do not suffer from this problem [10], [11], [12]. In this work, the parallel-prefix architecture proposed in [12] is utilized in order to achieve the fastest possible implementation. This architecture was derived by allowing the inverted reentering carry to recirculate at each existing prefix level. The design of these adders is briefly described as follows: At first, the carry-generate bits g_i, the carry-propagate bits p_i, and the half-sum bits h_i, for every i, $0 \leq i \leq n - 1$, are computed according to: $g_i = a_i \cdot b_i$, $p_i = a_i + b_i$, and $h_i = a_i \oplus b_i$, where $\cdot$, $+$, and $\oplus$ denote the logical AND, OR, and exclusive-OR operations, respectively. Then, using the bits g_i and p_i, the carries c_i, for $-1 \leq i \leq n - 2$, are computed in $\log_2 n$ prefix levels, according to the following relation:

$$(G_i, P_i) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \cdots \circ (g_0, p_0) \circ \overline{(g_{n-1}, p_{n-1}) \circ \cdots \circ (g_{i+1}, p_{i+1})},$$

with $c_i = G_i$. Finally, the sum bits s_i are derived using $s_i = h_i \oplus c_{i-1}$. By definition, $\overline{(g, p)}$ is equal to $(\overline{g}, p)$ and $\circ$ is the prefix operator defined as $(g, p) \circ (g', p') = (g + p \cdot g', p \cdot p')$.

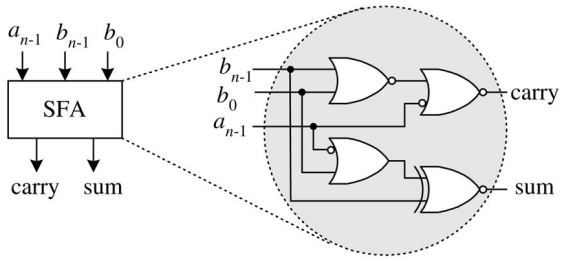

Fig. 1. Sample simplified-FA (SFA) implementation.

Additional simplifications are possible to the Dadda reduction tree. Consider the partial products $PP_0 = a_{n-1}b_0\, a_{n-2}b_0 \ldots a_1b_0\, a_0b_0$, $PP_n = A_{-1}$, and $PP_{n+1} = B_{-1}$. If these three partial products are driven to the same FA row of the array, then each FA can be simplified significantly. Fig. 1 presents a possible implementation of a block that accepts a_{n-1}, b_{n-1}, and b_0 and performs the addition of the bits $a_{n-1}b_0$, a_{n-1}, and b_{n-1}. The simplified FA is denoted as SFA. The FA of the same row that accepts a_0b_0, a_0, and b_0 can be further simplified to an HA. Furthermore, since C_{-1} is the all 0s vector, the row of FAs that accepts this operand can be simplified to a row of half-adders (HA).

Example 1. For a modulo 257 multiplier the derived set of partial products is shown in Fig. 2. Fig. 3 presents a numerical example illustrating the modulo partial-product reduction using the Dadda method. Every three terms are reduced to two, using an FA row, which is indicated by a box that surrounds them. The resulting sum and carry vectors are denoted as (S) and (C). The bold and underlined bits of each stage declare the carry bits of weight 2^8 that are complemented and added at the least-significant bit position. Additionally, Fig. 4 presents the attained FA-based implementation. Note that, in the first level of the tree, only HAs and SFAs have been used for reducing the delay. The circles at the carry output of an HA, FA, or an SFA denote the complement operation.

3 Comparisons

The multipliers designed according to the methods presented in [14], [15], and [18] require, apart from the partial-products reduction array, a final carry-propagate adder and a modulo correction step with a delay equal to an n-bit carry propagate adder. Thus, the proposed design and those of [16] and [17] that require only one n-bit carry-propagate addition are superior to these previous methods. Additionally, the authors of [16] and [17] have proven their superiority over [14] and [18]. Therefore, in this section, we compare the proposed (hereafter denoted block PROP)

$$
\begin{array}{lcccccccc}
PP_0 = & a_0b_7 & a_0b_6 & a_0b_5 & a_0b_4 & a_0b_3 & a_0b_2 & a_0b_3 & \overline{a_0b_0} \\
PP_1 = & a_1b_6 & a_1b_5 & a_1b_4 & a_1b_3 & a_1b_2 & a_1b_1 & a_1b_0 & \overline{a_1b_7} \\
PP_2 = & a_2b_5 & a_2b_4 & a_2b_3 & a_2b_2 & a_2b_1 & a_2b_0 & \overline{a_2b_7} & \overline{a_2b_6} \\
PP_3 = & a_3b_4 & a_3b_3 & a_3b_2 & a_3b_1 & a_3b_0 & \overline{a_3b_7} & \overline{a_3b_6} & \overline{a_3b_5} \\
PP_4 = & a_4b_3 & a_4b_2 & a_4b_1 & a_4b_0 & \overline{a_4b_7} & \overline{a_4b_6} & \overline{a_4b_5} & \overline{a_4b_4} \\
PP_5 = & a_5b_2 & a_5b_1 & a_5b_0 & \overline{a_5b_7} & \overline{a_5b_6} & \overline{a_5b_5} & \overline{a_5b_4} & \overline{a_5b_3} \\
PP_6 = & a_6b_1 & a_6b_0 & \overline{a_6b_7} & \overline{a_6b_6} & \overline{a_6b_5} & \overline{a_6b_4} & \overline{a_6b_3} & \overline{a_6b_2} \\
PP_7 = & a_7b_0 & \overline{a_7b_7} & \overline{a_7b_6} & \overline{a_7b_5} & \overline{a_7b_4} & \overline{a_7b_3} & \overline{a_7b_2} & \overline{a_7b_1} \\
PP_8 = & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
PP_9 = & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
PP_{10} = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

Fig. 2. The set of partial products for the proposed modulo $2^8 + 1$ multiplier.

multipliers against those of [16] (hereafter denoted block WANG) and [17] (hereafter denoted block MA), both qualitatively and quantitatively.

For our qualitative comparisons, we adopt the approximations of the unit-gate model [22], that is, we consider that all 2-input monotonic gates count as one gate equivalent for both area and delay, while a 2-input XOR or XNOR gate counts as two gate equivalents for both area and delay. We denote a Booth encoder by BE, a Booth selector block by BS, and a parallel modulo $2^n + 1$ adder by PA_n. The area of a block Y will be denoted A_Y and its execution latency as T_Y. The area and delay in equivalent gates of the components used in the comparisons are shown in Table 1.

In the proposed multipliers, $n + 3$ partial products are required. The three of them are bits from the input operands, which are added using the SFA cells, while one of them is the all zeros vector. The rest of the partial products are produced by $n(n-1)$ AND or NAND gates. These partial products are then reduced to two by the use of a Dadda tree. The depth in FA stages of a Dadda tree, denoted $D(k)$, is a function of its number of operands and is listed in Table 2 for all practical values of k. Each of the n columns of the tree, except the least significant one, is composed of $n-1$ FAs, 1 SFA, and 1 HA. The least significant slice is composed of $n-1$ FAs and 2 HAs. Therefore, the total area of the Dadda tree required by the proposed multipliers is $A_{DT} = n(n-1)A_{FA} + (n-1)A_{SFA} + (n+1)A_{HA}$, while its execution delay is $T_{DT} = D(n+3)T_{FA}$. As exemplified in the previous section, in several cases, it is possible to arrange the first level of the Dadda tree so that it is composed only of SFAs or of SFAs and HAs. This can be achieved in the cases where $(n+2)$ or $(n+1)$ is a Dadda number, i.e., when $n = 4, 5, 7, 8, 11, 12, 17, 18, 26, 27, \ldots$. In these cases, the execution delay of the Dadda tree is $T_{DT} = (D(n+3) - 1)T_{FA} + T_{HA}$. Taking into account the approximations of the unit gate model, we get that

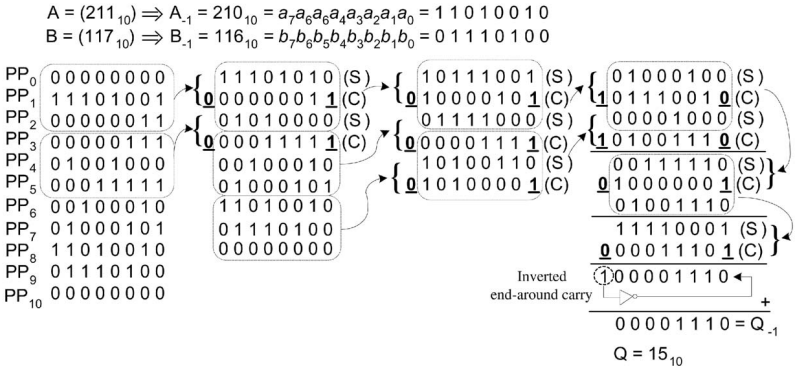

Fig. 3. Numerical example in the case of the proposed modulo $2^8 + 1$ multiplier.

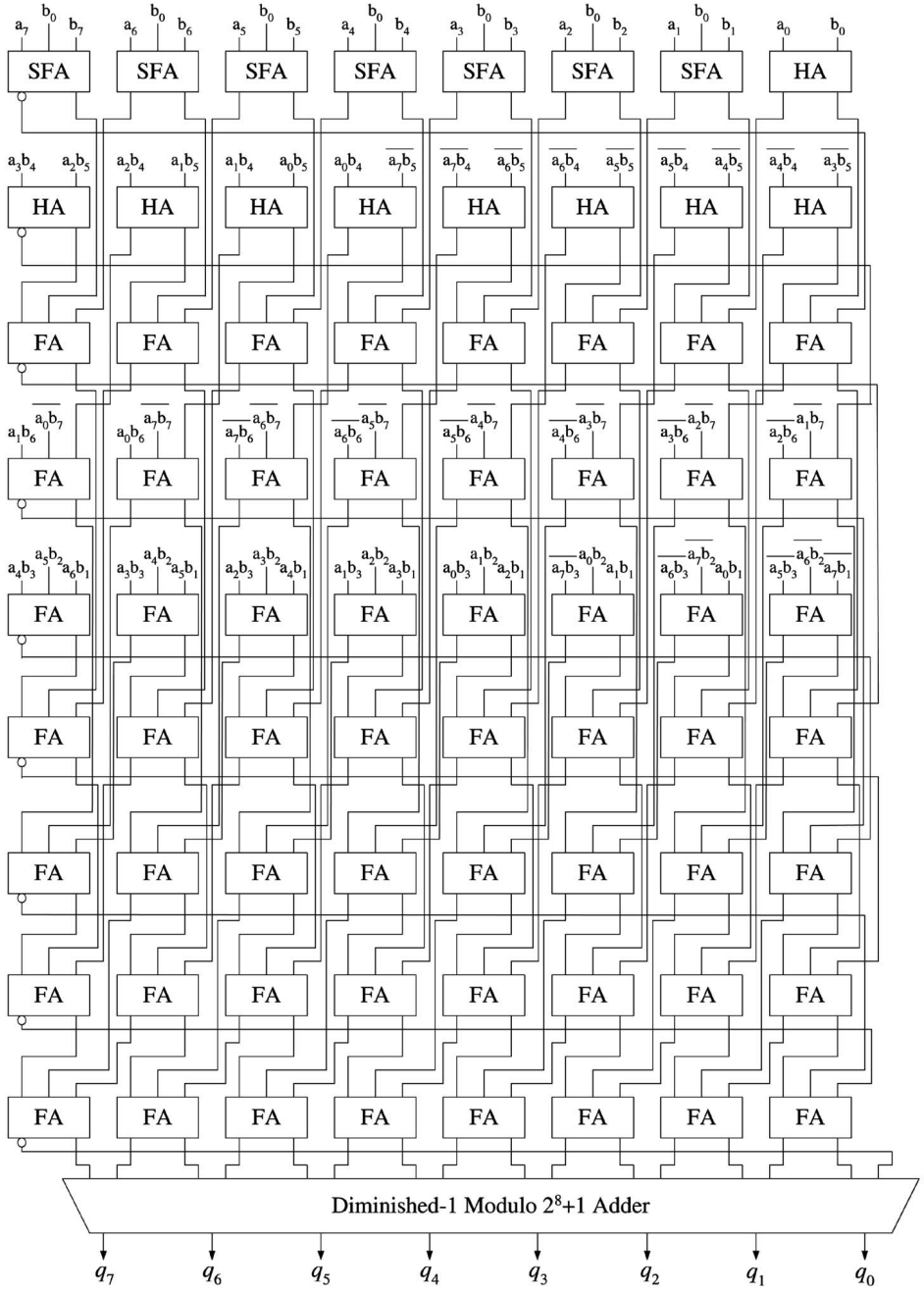

Fig. 4. The proposed modulo $2^8 + 1$ multiplier.

$$A_{PROP} = n^2 + A_{DT} + A_{PA_n} = 8n^2 + \frac{9}{2}n\log_2 n + \frac{1}{2}n + 4 \text{ equivalent gates}, \quad (17)$$

$$T_{PROP} = 1 + T_{DT} + T_{PA_n} = \begin{cases} 4D(n+3) + 2\log_2 n + 2, & \text{if } n = 4,5,7,8,11,12,17,18,\ldots \\ 4D(n+3) + 2\log_2 n + 4, & \text{otherwise.} \end{cases} \quad (18)$$

TABLE 1
Area and Delay of the Basic Components in Equivalent Gates

	BE	BS	FA	SFA	HA	PA_n
Area	5	5	7	5	3	$\frac{9}{2}n\log_2 n + \frac{1}{2}n + 6$
Delay	3	4	4	3	2	$2\log_2 n + 3$

The multipliers proposed in [16] follow a similar structure as the proposed ones. However, the following should be noted:

- $n+1$ partial products are utilized. Out of them, $n-1$ are produced using two input AND gates. However, these AND gates require that one of their input operands be inverted. One partial product is produced by the use of $2 \rightarrow 1$ multiplexors. We consider that a multiplexor has the same complexity as an XOR gate. The final partial product is the inverse of the number of zeros in the $n-1$ bits from

TABLE 2
FA Stages in a k Operand Dadda Tree

k	4	5...6	7...9	10...13	14...19	20...28	29...42	43...63	64...94
$D(k)$	2	3	4	5	6	7	8	9	10

TABLE 3
Area and Delay in Equivalent Gates

n	A_{WANG} [16]	T_{WANG} [16]	A_{MA} [17]	T_{MA} [17]	A_{PROP}	T_{PROP}
4	167	25	179	28	170	24
8	634	33	586	38	628	28
12	1,373	39	1,213	44	1,356	34
16	2,379	43	2,019	44	2,348	36
20	3,648	49	3,045	50	3,603	42
24	5,178	49	4,261	50	5,120	42
28	6,969	49	5,674	54	6,896	46
32	9,020	53	7,268	54	8,932	46

b_1 to b_{n-1}. This number is provided by an $n-1$ bits to $\lceil \log_2(n-1) \rceil$ counter (denoted by CNT).

- In [16], it is proposed to reduce the partial products in two final summands by the use of a Wallace tree. In our comparisons, we assume that this reduction is performed by a Dadda tree. The latter has the same time complexity while it, in parallel, offers reduced area complexity.
- The two final summands are added in a modulo 2^n+1 parallel adder with a carry input set to 1. Since such a block is not available in the literature, we assume that this is implemented by an HA stage, followed by a modulo 2^n+1 parallel adder.

The area requirements of the multipliers proposed in [16] are:

- $n(n-1)$ AND and n XOR gates for forming the n partial products.
- $(n-1)-\lceil \log_2(n-1) \rceil$ FAs for the CNT block that forms the last partial product.
- $n(n-1)$ FAs for the Dadda tree and n HAs for producing the two final summands.
- A modulo 2^n+1 adder PA_n.

Taking into account the approximations of the unit-gate model, it follows that

$$A_{WANG} = 8n^2 + \frac{9}{2} n \log_2 n + \frac{9}{2} n - 7\lceil \log_2(n-1) \rceil - 1. \tag{19}$$

Considering the execution delay, one must note that:

- The terms of the $n-1$ partial products require more than a single gate delay to be produced since each is the AND of a normal input bit with the other inverted.
- The multiplexors impose an extra delay for the derivation of this specific partial product against the rest. In order to compensate for this extra delay, the output of the multiplexors should be driven to the second or to subsequent stages of the Dadda tree. However, this is not possible, when $n+1$ is a Dadda number or, equivalently, when $n = 5, 8, 12, 18, 27, 41, 62, 93 \ldots$.
- Finally, the partial product produced by the CNT may also not be ready when needed for a minimum depth Dadda tree. Because of this, we cannot provide a closed form equation for T_{WANG}. In our estimation, we consider that the CNT is designed according to [23].

The multipliers proposed in [17] use Booth recoding to reduce the number of partial products that should be added. In the following, we consider that n is even. The number of derived partial products in [17] is $\frac{n}{2}+1$, each $(n+1)$-bits wide. One of the partial products is a constant, whereas the rest are derived using a Booth encoder for each overlapping triplet of the multiplier and $n+1$ Booth selector blocks. In [17], it is proposed that these partial products are reduced into a carry and sum vector using a Carry-Save Adder (CSA) Array. In the following, we consider that this is performed by a Dadda tree to reduce the delay. The number of FA stages in the Dadda tree is $D(\frac{n}{2}+1)$, whereas the number of FAs and HAs required is $\frac{1}{2}n(n-1) - 2\lfloor \log_2(\frac{n}{2}+1) \rfloor$ and $\frac{n}{2}$, respectively. The sum and carry vectors produced are then fed into two cascaded modulo CSA stages, each contributing T_{FA} of execution delay. The first stage, because of the constants in the high order bits of the sum and carry vectors, can be implemented by 1 HA and $\lceil \log_2(\frac{n}{2}) \rceil$ FAs, whereas the second requires n FAs. The two resulting vectors need to be added in a modulo 2^n+1 parallel adder with a carry input set to 1, as in the case of the multipliers proposed in [16]. Also, in this case, we assume that this is implemented by an HA stage, followed by a modulo 2^n+1 parallel adder. According to the above analysis, we have that, for even values of n:

$$\begin{aligned} A_{MA} &= \frac{n}{2} A_{BE} + \frac{n}{2}(n+1) A_{BS} \\ &\quad + \left(\frac{1}{2} n(n-1) - 2\left\lfloor \log_2\left(\frac{n}{2}+1\right) \right\rfloor + \left\lceil \log_2\left(\frac{n}{2}\right) \right\rceil + n \right) A_{FA} \\ &\quad + \left(\frac{n}{2} + 1 + n \right) A_{HA} + A_{PA_n} \\ &= 6n^2 + \frac{9}{2} n \log_2 n + \frac{27}{2} n + 7\left\lceil \log_2 \frac{n}{2} \right\rceil - 14\left\lfloor \log_2\left(\frac{n}{2}+1\right) \right\rfloor \end{aligned} \tag{20}$$

$$\begin{aligned} T_{MA} &= T_{BE} + T_{BS} + D\left(\frac{n}{2}+1\right) T_{FA} + 2T_{FA} + T_{HA} + T_{PA_n} \\ &= 20 + 4D\left(\frac{n}{2}+1\right) + 2\log_2 n. \end{aligned} \tag{21}$$

Taking into account the area estimates of (17), (18), (19), and the analysis presented earlier for the delay T_{WANG}, (20) and (21), we present in Table 3 the delay and area requirements of the multipliers under consideration for several values of n. The proposed multipliers offer significant savings in execution time compared to either the multipliers proposed in [16] or in [17]. The

TABLE 4
Area (μm^2) and Delay (ns) Results of the Diminished-1 Modulo $2^n + 1$ Multipliers

n	Wang [16]		Ma [17]		Proposed	
	Area	Delay	Area	Delay	Area	Delay
4	3,939	1.51	3,134	1.64	3,724	1.34
8	10,427	2.19	8,830	2.32	9,685	1.98
16	38,619	2.94	29,856	3.07	36,541	2.65
32	126,792	3.91	103,287	4.06	118,552	3.63

TABLE 5
Area (μm^2) and Power (mW) Results for the Diminshed-1 Modulo $2^n + 1$ Multipliers

n	Delay	Wang [16]		Ma [17]		Proposed	
		Area	Power	Area	Power	Area	Power
4	1.64	2,987	0.92	3,134	1.05	2,865	0.84
8	2.32	8,896	3.32	8,830	4.01	8,728	2.85
16	3.07	32,785	9.19	29,856	10.37	30,347	7.62
32	4.06	104,568	18.35	103,287	21.46	98,621	16.12

proposed multipliers are also more area efficient than the multipliers in [16] for $n > 4$. Finally, considering as a metric the area $\times$ time product, the proposed multipliers are more efficient than the multipliers proposed in [17] for $n < 24$.

Quantitative comparison results are obtained by implementing the different multiplier architectures into a 0.18μm CMOS standard cell library. At first, a program was written in C++ that generates structural Verilog descriptions for the proposed and the multipliers proposed in [16] and [17]. We used this program to generate Verilog models for multipliers with operand sizes of 4, 8, 16, and 32 bits. Each design, after performing extensive simulations that verified its correctness, was synthesized and optimized recursively for minimum delay, with Synopsys Design Compiler using the UMC 0.18μm CMOS standard cell library (five metal layers), under typical conditions (1.8Volt, 25° C). Then, the derived netlists and the design constraints were passed to Cadence Silicon Ensemble to perform the final placement and routing of the design. All design constraints, such as output load, max fanout, and floorplan initialization information, were held constant for each architecture. Final timing analysis was performed using PrimeTime of Synopsys toolset after all RC parasitic information were extracted from the layout and back-annotated to the gate-level netlist. Table 4 shows the obtained area and delay results. The reported area measurements are performed in the final layout and include both cell and interconnect area. The simulation data indicate that the proposed multipliers offer delay savings between 7 percent and 11 percent over the multipliers in [16] and between 10 percent and 18 percent over the multipliers in [17]. Additionally, in all cases, they are more area efficient than the multipliers of [16] by 6 percent on average.

In order to measure power consumption, all designs were optimized targeting a delay equal to the minimum delay of the Booth modulo $2^n + 1$ multipliers proposed by Ma in [17]. The resulting netlists were placed and routed and the parasitics were extracted. All gathered design data were passed to PrimePower of Synopsys and power was estimated after the application of $5,000$ random vectors. Experimental results, shown in Table 5, indicate that the proposed multipliers in the majority of the cases require the smallest implementation area, while their power consumption is less than the multipliers of [16] and [17] by 13 percent and 23 percent on average.

4 Conclusions

In this paper, we have proposed a new algorithm for designing diminished-1 modulo $2^n + 1$ multipliers. The proposed multipliers offer significant savings in propagation delay compared to the already known ones and they are more area and power efficient for less strict delay constraints.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. G. Dimitrakopoulos has been supported by the "D. Maritsas" Graduate Scholarship.

References

[1] F. Taylor, "A Single Modulus ALU for Signal Processing," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 33, pp. 1302-1315, 1985.
[2] E. DiClaudio et al., "Fast Combinatorial RNS Processors for DSP Applications," *IEEE Trans. Computers*, vol. 44, pp. 624-633, 1995.
[3] J. Ramirez et al., "RNS-Enabled Digital Signal Processor Design," *IEE Electronics Letters*, vol. 38, no. 6, pp. 266-268, 2002.
[4] R. Chaves and L. Sousa, "RDSP: A RISC DSP Based on Residue Number System," *Proc. Euromicro Symp. Digital Systems Design*, pp. 128-135, Sept. 2003.
[5] L.M. Leibowitz, "A Simplified Binary Arithmetic for the Fermat Number Transform," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 24, pp. 356-359, 1976.
[6] T.K. Truong et al., "Techniques for Computing the Discrete Fourier Transform Using the Quadratic Residue Fermat Number Systems," *IEEE Trans. Computers*, vol. 35, pp. 1008-1012, 1986.
[7] M. Benaissa et al., "Diminished-1 Multiplier for a Fast Convolver and Correlator Using the Fermat Number Transform," *IEE Proc. G*, vol. 135, pp. 187-193, 1988.
[8] S. Sunder at al., "Area-Efficient Diminished-1 Multiplier for Fermat Number-Theoretic Transform," *IEE Proc. G*, vol. 140, pp. 211-215, 1993.
[9] R. Zimmermann et al., "A 177 Mb/s VLSI Implementation of the International Data Encryption Algorithm," *IEEE J. Solid-State Circuits*, vol. 29, no. 3, pp. 303-307, 1994.
[10] R. Zimmerman, "Efficient VLSI Implementation of Modulo $(2^n \pm 1)$ Addition and Multiplication," *Proc. IEEE Symp. Computer Arithmetic*, pp. 158-167, Apr. 1999.
[11] C. Efstathiou et al., "Modulo $2^n \pm 1$ Adder Design Using Select-Prefix Blocks," *IEEE Trans. Computers*, vol. 52, pp. 1399-1406, 2003.
[12] H.T. Vergos, C. Efstathiou, and D. Nikolos, "Diminished-One Modulo $2^n + 1$ Adder Design," *IEEE Trans. Computers*, vol. 51, pp. 1389-1399, 2002.
[13] S.J. Piestrak, "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders," *IEEE Trans. Computers*, vol. 43, pp. 68-77, 1994.
[14] A.A. Hiasat, "A Memoryless $\mathrm{mod}(2^n \pm 1)$ Residue Multiplier," *Electronics Letters*, vol. 28, no. 3, pp. 314-315, 1992.
[15] A. Wrzyszcz and D. Milford, "A New Modulo $2^a + 1$ Multiplier," *Proc. Int'l Conf. Computer Design*, pp. 614-617, 1993.
[16] Z. Wang, G.A. Jullien, and W.C. Miller, "An Efficient Tree Architecture for Modulo $2^n + 1$ Multiplication," *J. VLSI Signal Processing*, vol. 14, pp. 241-248, 1996.
[17] Y. Ma, "A Simplified Architecture for Modulo $(2^n + 1)$ Multiplication," *IEEE Trans. Computers*, vol. 47, no. 3, pp. 333-337, Mar. 1998.
[18] A.V. Curiger et al., "Regular VLSI Architectures for Multiplication Modulo $(2^n + 1)$," *IEEE J. Solid-State Circuits*, vol. 26, no. 7, pp. 990-994, 1991.
[19] V. Paliouras, A. Skavantzos, and T. Stouraitis, "Multi-Voltage Low Power Convolvers Using the Polynomial Residue Number System," *Proc. ACM Great Lakes Symp. VLSI*, pp. 7-11, 2002.
[20] A. Hammalainen, M. Tommiska, and J. Skytta, "6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm," *Lecture Notes in Computer Science*, vol. 2438, pp. 760-769, 2002.
[21] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, 1965.
[22] A. Tyagi, "A Reduced-Area Scheme for Carry-Select Adders," *IEEE Trans. Computers*, vol. 42, no. 10, pp. 1163-1170, Oct. 1993.
[23] E.E. Swartzlander, "Parallel Counters," *IEEE Trans. Computers*, vol. 22, pp. 1021-1024, 1973.