A Family of Modulo $(2^n + 1)$ Multipliers

Jean-Luc Beuchat

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

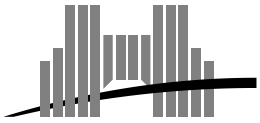
A Family of Modulo $(2^n + 1)$ *Multipliers*

Jean-Luc Beuchat

September 2004

Research Report N° RR2004–39

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



A Family of Modulo $(2^n + 1)$ Multipliers

Jean-Luc Beuchat

September 2004

Abstract

In this paper, we first describe a novel modulo $(2^n + 1)$ addition algorithm suited to FPGA and ASIC implementations, and discuss several architectures of multioperand modulo (2^n+1) adders. Then, we propose three implementations of a modulo (2^n+1) multiplication algorithm based on a paper by A. Wrzyszcz and D. Milford. The first operator is based on an $n \times n$ multiplication and a subsequent modulo $(2^n + 1)$ correction, and takes advantage of the arithmetic logic embedded in Spartan or Virtex FPGAs. The second operator computes a sum of modulo-reduced partial products by means of a multioperand modulo $(2^n + 1)$ adder. Then, radix-4 modified Booth recoding reduces the number of partial products, while making their generation more complex. Finally, we provide a comparison of this family of algorithms with existing solutions.

Keywords: Modular arithmetic, modulo $(2^n + 1)$ addition, modulo $(2^n + 1)$ multiplication, FPGA implementation

Résumé

Dans cet article, nous présentons un nouvel algorithme d'addition modulo $(2^n + 1)$ et discutons son implantation pour des circuits ASIC ou FPGA. Nous proposons ensuite trois implantations d'un algorithme de multiplication modulo $(2^n + 1)$ inspiré d'un article de Wrzyszcz et Milford. Le premier opérateur est constitué d'un multiplieur $n \times n$ et d'une unité de calcul effectuant une réduction modulaire. Il tire ainsi parti de la logique destinée aux opérations arithmétiques disponible dans les familles de FPGA Spartan ou Virtex. Le second opérateur calcule la somme de produits partiels réduits au moyen d'un additionneur de plusieurs opérandes modulo $(2^n + 1)$. Le recodage modifié de Booth permet ensuite de réduire le nombre de produits partiels, tout en compliquant leur génération. Finalement, nous proposons une comparaison entre cette famille d'algorithmes et des solutions existantes.

Mots-clés: Arithmétique modulaire, addition modulo $(2^n + 1)$, multiplication modulo $(2^n + 1)$, implantation sur FPGA

1 Introduction

This paper is devoted to the study of modulo (2^n+1) multipliers for both ASIC and FPGA designs. Such operators play a crucial role in several domains:

- *Cryptography.* The security of the IDEA block cipher [11] relies on combining operations from three different groups of 2^{16} elements. The group operations are: modulo 2^{16} addition, bitwise exclusive OR, and modulo $(2^{16}+1)$ multiplication. Numerous papers emphasized that the encryption rate mainly depends on the architecture of the modulo $(2^{16}+1)$ multiplier (see for instance [24, 16, 4, 8, 9, 14]).
- *Residue number system (RNS).* The Chinese remainder theorem provides a method to perform the RNS to weighted number system conversion. Researchers proposed several sets of moduli allowing better hardware implementations, such as $\{2^n-1, 2^n, 2^n+1\}$ (see for instance [7, 15, 17]). Thus, digital signal processing systems taking advantage of this 3-moduli set require efficient modulo $(2^n \pm 1)$ adders and multipliers.
- Other applications include the Fermat number transform [3, 12, 1], pseudorandom number generation, or error correcting codes [5].

The algorithms proposed in this paper are based on the modulo (2^n+1) addition of (modulo-reduced) partial products, whose generation follows from properties of modulo (2^n+1) arithmetic (Section 2). Therefore, we first study several (multioperand) modulo (2^n+1) addition algorithms, and discuss their FPGA and ASIC implementations (Section 3). Then, we describe a modulo (2^n+1) multiplication algorithm based on a paper by A. Wrzyszcz and D. Milford [21], and discuss three implementations taking advantage of our adder structures (Section 4). Finally, Section 5 provides a comparison on FPGA of our algorithms with existing solutions. Lemma proofs are provided in appendices. The notation adopted throughout this paper is summarized in Table 1.

Table 1: Notation adopted throughout the paper.

Notation	Explanation
CRA	Carry-Ripple Adder.
CSA	Carry-Save Adder.
FA	Full-Adder.
LUT	Look-Up Table.
$X_{m-1:0}$	m-digit radix-2 number; when the length is obvious, we use X as a shorthand.
x_j	Single digit of X at position j.
$X_{h:l}$	$(h-l+1)$-digit number defined by $\sum_{i=0}^{h-l} x_{l+i}2^i$.
$\langle X \rangle_M$	Remainder: $X - M\lfloor \frac{X}{M} \rfloor$.
X'	Diminished-one representation of X.
$X \wedge Y$	Bitwise AND of two bit strings X and Y of the same length.
$X \vee Y$	Bitwise OR of two bit strings X and Y of the same length.
$X \oplus Y$	Bitwise exclusive-OR of two bit strings X and Y of the same length.
$\overline{X_{m-1:0}}$	Logical negation of the m-bit number $X_{m-1:0}$: $\sum_{i=0}^{m-1}(1-x_i)2^i = \sum_{i=0}^{m-1}\bar{x}_i 2^i$.

2 Background

The algorithms studied in this paper are based on two well-known properties of modulo (2^n+1) arithmetic (see for instance [10, 21]).

Property 1. $\langle 2^n \rangle_{2^n+1} = \langle -1 \rangle_{2^n+1}$.

Property 2. *Let X be an n-bit unsigned number. Then,*

$$\langle 2^i X \rangle_{2^n+1} = \langle 2^i X_{n-i-1:0} + \overline{X_{n-1:n-i}} - 2^i + 1 \rangle_{2^n+1},$$

and

$$\langle -2^i X \rangle_{2^n+1} = \langle 2^i \overline{X_{n-i-1:0}} + X_{n-1:n-i} + 2^i + 1 \rangle_{2^n+1}.$$

Proof. Property 1 allows to demonstrate the correctness of both equalities:

$$\begin{aligned}
\langle 2^i X \rangle_{2^n+1} &= \langle 2^n X_{n-1:n-i} + 2^i X_{n-i-1:0} \rangle_{2^n+1} \\
&= \langle -X_{n-1:n-i} + 2^i X_{n-i-1:0} \rangle_{2^n+1} \\
&= \langle 2^i X_{n-i-1:0} + 2^i - X_{n-1:n-i} - 2^i \rangle_{2^n+1} \\
&= \left\langle 2^i X_{n-i-1:0} + 1 \right. \\
&\quad \left. + \sum_{j=0}^{i-1} (1 - x_{n-i+j}) 2^j - 2^i \right\rangle_{2^n+1} \\
&= \langle 2^i X_{n-i-1:0} + \overline{X_{n-1:n-i}} - 2^i + 1 \rangle_{2^n+1},
\end{aligned}$$

and

$$\begin{aligned}
\langle -2^i X \rangle_{2^n+1} &= \langle -2^n X_{n-1:n-i} - 2^i X_{n-i-1:0} \rangle_{2^n+1} \\
&= \langle (2^n - 2^i X_{n-i-1:0}) \\
&\quad + (X_{n-1:n-i} + 1) \rangle_{2^n+1} \\
&= \left\langle \left(\sum_{j=0}^{n-i-1} (1 - x_j) 2^{i+j} + \sum_{j=0}^{i-1} 2^j + 1 \right) \right. \\
&\quad \left. + (X_{n-1:n-i} + 1) \right\rangle_{2^n+1} \\
&= \langle 2^i \overline{X_{n-i-1:0}} + X_{n-1:n-i} + 2^i + 1 \rangle_{2^n+1}.
\end{aligned}$$

□

3 Modulo (2^n+1) Addition

Modulo (2^n+1) addition is efficiently performed in the diminished-one number system, where a number X is

represented by $X' = \langle X-1\rangle_{2^n+1}$. The sum $S = \langle S'+1\rangle_{2^n+1}$ of two operands X and Y can be written:

$$\langle S'+1\rangle_{2^n+1} = \left\langle \langle X'+1\rangle_{2^n+1} + \langle Y'+1\rangle_{2^n+1} \right\rangle_{2^n+1}.$$

Consequently, modulo (2^n+1) addition in the diminished-one number system is defined by:

$$\langle S'\rangle_{2^n+1} = \langle X'+Y'+1\rangle_{2^n+1}. \quad (1)$$

If the input pattern $X' = Y' = 2^n$ never occurs, modulo (2^n+1) addition can be carried out according to the following algorithm, studied for instance by R. Zimmermann in [22, 23]:

$$\langle X'+Y'+1\rangle_{2^n+1} = T_{n-1:0} + \bar{t}_n, \quad (2)$$

where $T = X' + Y'$. A straightforward modification to this algorithm allows to handle the case where both operands are equal to 2^n (see for instance [9]):

$$\langle X'+Y'+1\rangle_{2^n+1} = \begin{cases} 2^n & \text{if } X' = Y' = 2^n, \\ T_{n-1:0} + \bar{t}_n & \text{otherwise.} \end{cases} \quad (3)$$

The hardware implementation of such an equation requires a multiplexer which selects the correct result according to $x'_n y'_n$. The following result leads to a smaller operator by getting rid of the multiplexer[1].

Lemma 1. *Consider two numbers $X', Y' \in \{0,\ldots,2^n\}$, and the $(n+1)$-bit sum $U_{n:0} = X'_{n-1:0} + Y'_{n-1:0}$. Then,*

$$\langle X'+Y'+1\rangle_{2^n+1} = (x'_n y'_n 2^n + U_{n-1:0}) + \overline{x'_n \vee y'_n \vee u_n}. \quad (4)$$

The implementation of this modulo (2^n+1) addition scheme depends on the target technology. R. Zimmermann's synthesis results point out that (2) is efficiently implemented in cell-based VLSI technologies by an end-around-carry parallel-prefix adder, whose output carry c_{out} is connected to the input carry c_{in} [22, 23]. The same approach can be applied to the algorithm defined by (4) (Figure 1a). In the following, we adopt the notation introduced in [6] to describe prefix networks. A parallel prefix algorithm (e.g. Sklansky or Brent-Kung) computes the generate/propagate signals $(g_{(i,0)}, p_{(i,0)})$, $0 \le i \le n-1$, from $X'_{n-1:0}$ and $Y'_{n-1:0}$. Since $u_n = g_{(n-1,0)}$, we have to add $c_0 = \overline{x'_n \vee y'_n \vee g_{(n-1,0)}}$ to this intermediate result. We apply the late input carry scheme [22] which requires an additional prefix level.

The direct connection of $\overline{x'_n \vee y'_n \vee u_n}$ to c_{in} turns the CRAs available in modern FPGAs into asynchronous sequential circuits. This problem is solved by performing the modulo (2^n+1) addition in two steps:

- a first CRA computes the $(n+1)$-bit number $U = X'_{n-1:0} + Y'_{n-1:0}$;
- then a second CRA increments $x'_n y'_n 2^n + U_{n-1:0}$ by $\overline{x'_n \vee y'_n \vee u_n}$ (Figure 1b).

When their inputs are radix-2 numbers in $\{0,\ldots,2^n\}$, the operators described in this Section return the sum increased by one modulo (2^n+1). Nevertheless, this property can be dealt with in many applications, such as modulo (2^n+1) multiplication. Note finally that a diminished-one to radix-2 integer conversion scheme [12] is obtained by substituting 0 for Y' in (4):

$$\langle X'+1\rangle_{2^n+1} = X'_{n-1:0} + \bar{x}'_n. \quad (5)$$

3.1 Modulo (2^n+1) Subtraction

The modulo (2^n+1) subtraction of two radix-2 numbers A and B $(0 \le A, B < 2^n)$ is defined by:

$$\begin{aligned} &\langle A - B\rangle_{2^n+1} \\ &\quad = \langle (2^n + A) - B + 1\rangle_{2^n+1} \\ &\quad = \begin{cases} (2^n + A) - B + 1 & \text{if } (2^n + A) - B < 2^n, \\ A - B & \text{otherwise.} \end{cases} \end{aligned} \quad (6)$$

Consider the $(n+1)$-bit number $D = (2^n + A) - B$. Then,

$$D_{n-1:0} = \begin{cases} (2^n + A) - B & \text{if } D < 2^n, \\ A - B & \text{otherwise,} \end{cases}$$

and

$$\langle A - B\rangle_{2^n+1} = D_{n-1:0} + \bar{d}_n. \quad (7)$$

This operation plays for instance an important role in the implementation of multioperand modulo (2^n+1) adders on FPGAs embedding dedicated carry logic. A specific subtraction is required to convert an $(n+1)$-bit number $X \in \{0,\ldots,2^n\}$ into the diminished-one number system. Let $D = \langle X-1\rangle_{2^n+1}$. Then

$$X' = \langle X-1\rangle_{2^n+1} = d_n + \langle D\rangle_{2^n}. \quad (8)$$

To prove the correctness of this algorithm, it suffices to note that

$$X' = \begin{cases} 1 + (2^n - 1) = 2^n & \text{if } X = 0, \\ 0 + X - 1 = X - 1 & \text{otherwise.} \end{cases}$$

Another solution consists in computing the sum of X and $2^n - 1$ with the first modulo (2^n+1) adder described in this paper [12].

3.2 Multioperand Modulo (2^n+1) Addition

We deduce from (1) that the modulo (2^n+1) addition of m operands $X^{(i)}$ in the diminished-one number system is defined by:

$$\langle S'\rangle_{2^n+1} = \left\langle m - 1 + \sum_{i=0}^{m-1} X^{(i)'} \right\rangle_{2^n+1}.$$

[1] We proposed a first version of this algorithm in [2].

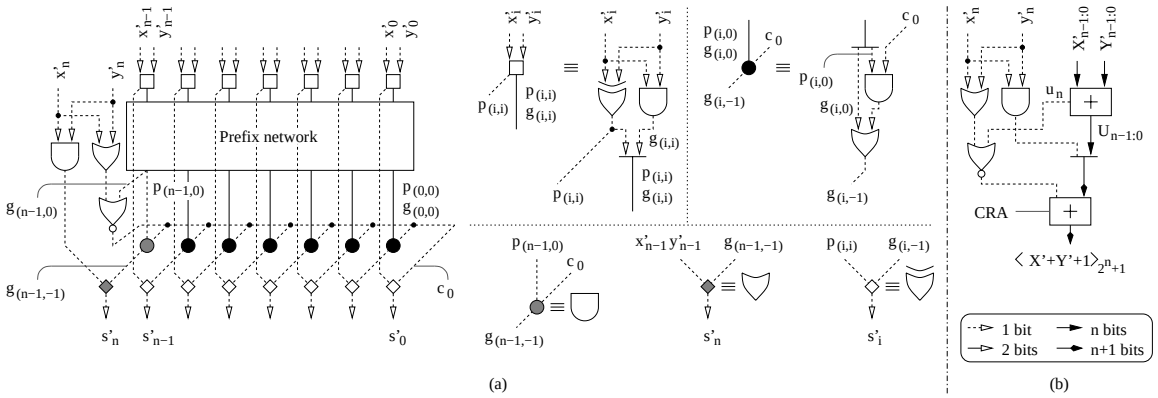


Figure 1: Modulo (2^n+1) adders in the diminished-one number system.

Again, the implementation of this equation depends on the target technology. CSA-based architectures are obviously well suited to ASIC circuits [13, 23]. When all inputs belong to $\{0,\ldots,2^n-1\}$, R. Zimmermann proposed to compute a sum-bit vector $A_{n-1:0}$ and a carry-bit vector $B_{n-1:0}$ by means of an m-operand end-around-carry CSA [23]. This operator carries out $(m-2)$ modulo (2^n+1) additions in the diminished-one number system. Consequently,

$$\langle A+B\rangle_{2^n+1} = \left\langle m-2+\sum_{i=0}^{m-1} X^{(i)\prime}\right\rangle_{2^n+1}.$$

Then, a parallel-prefix modulo (2^n+1) adder performs the carry-save to integer conversion, while adding an extra '1':

$$\langle S'\rangle_{2^n+1} = \langle A+B+1\rangle_{2^n+1}.$$

Since A and B belong to $\{0,\ldots,2^n-1\}$, we deduce from (4) that $\langle A+B+1\rangle_{2^n+1} = U_{n-1:0} + \overline{u_n}$, where $U_{n:0} = A+B$. We can therefore simplify the operator illustrated in Figure 1a as follows: $s'_n = g_{(n-1,-1)}$ and $c_0 = \overline{g_{(n-1,0)}}$.

Unfortunately, a CSA does not take advantage of the dedicated carry logic available in several FPGA families. On Xilinx Virtex or Spartan devices, two LUTs are required to generate the sum bit and the carry bit of each FA cell of a CSA, whereas a single LUT and arithmetic logic implement an FA cell of a CRA. Therefore, Hämäläinen *et al.* [9] suggested to carry out the multioperand modular addition by means of a tree of 2-input CRA-based modulo (2^n+1) adders. If $1+\lceil\log_2 m\rceil \le n$, a third solution consists in computing the $(n+1+\lceil\log_2 m\rceil)$-bit number $A = m-1+\sum_{i=0}^{m-1} X^{(i)\prime}$ with a tree of $(m-1)$ CRAs whose input carry is set to 1, and in performing a final correction by means of the modulo (2^n+1) subtracter studied in Section 3.1:

$$\langle S'+1\rangle_{2^n+1} = \left\langle A_{n-1:0} - A_{n+\lceil\log_2 m\rceil:n}\right\rangle_{2^n+1}.$$

The hypothesis on m guarantees that $A_{n+\lceil\log_2 m\rceil:n}$ is an n-bit number.

4 Modulo (2^n+1) Multiplication

In this paper, we start from the following definition of modulo (2^n+1) multiplication:

$$\begin{aligned}\langle XY\rangle_{2^n+1} &= \langle (x_n 2^n + X_{n-1:0})\cdot(y_n 2^n + Y_{n-1:0})\rangle_{2^n+1}\\ &= \langle x_n y_n 2^{2n} + (y_n X_{n-1:0} + x_n Y_{n-1:0})2^n\\ &\quad + X_{n-1:0}Y_{n-1:0}\rangle_{2^n+1}. \end{aligned} \tag{9}$$

Then, we apply a clever idea which was, to our best knowledge, initially proposed by A. Wrzyszcz and D. Milford [21], and which allows to replace an addition by a bitwise OR in (9). Since $0 \le X, Y \le 2^n$, we have $X_{n-1:0} = \bar{x}_n X_{n-1:0}$, $Y_{n-1:0} = \bar{y}_n Y_{n-1:0}$, and

$$\begin{aligned} y_n X_{n-1:0} + x_n Y_{n-1:0} &= y_n X_{n-1:0} \vee x_n Y_{n-1:0}\\ &= y_n \bar{x}_n X_{n-1:0} \vee x_n \bar{y}_n Y_{n-1:0}\\ &= (x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0}). \end{aligned}$$

Substituting the above equality in (9) yields:

$$\begin{aligned}\langle XY\rangle_{2^n+1} &= \langle X_{n-1:0}Y_{n-1:0}\\ &\quad - ((x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0}))\\ &\quad + x_n y_n\rangle_{2^n+1}, \end{aligned} \tag{10}$$

$$\begin{aligned}&= \langle X_{n-1:0}Y_{n-1:0}\\ &\quad + \overline{(x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0})}\\ &\quad + x_n y_n + 2\rangle_{2^n+1}. \end{aligned} \tag{11}$$

Broadly speaking, there are four methodologies to implement such an algorithm [5]:

- $\langle X_{n-1:0}Y_{n-1:0}\rangle_{2^n+1}$ can be computed by means of look-up tables. The major drawback of this approach lies in the exponential growth of the required memory size ($n2^{2n}$ bits). A slightly better algorithm, known as *quarter-squared multiplier* (see for instance [20]), is based on the following idea: $X_{n-1:0}Y_{n-1:0} = (S^+)^2 - (S^-)^2$, where $S^+ = (X_{n-1:0}+Y_{n-1:0})/2$ and $S^- = (X_{n-1:0}-Y_{n-1:0})/2$. In modular form, this equation becomes:

$$\langle X_{n-1:0}Y_{n-1:0}\rangle_{2^n+1} = \left\langle \varphi(S^+) - \varphi(S^-)\right\rangle_{2^n+1},$$

where $\varphi(A) = \langle A^2 \rangle_{2^n+1}$ is stored in a table. This method reduces the memory growth to $n(2^{n+1}+2^n)$.

If (2^n+1) is a prime, the mathematical properties associated with the Galois field $\mathbb{F}_{2^n+1}$ allow to design another table-based algorithm [18]. All the nonzero elements of $\mathbb{F}_{2^n+1}$ can be generated using a primitive element α. The mapping is given by $X_{n-1:0} = \alpha^{i_x}$ and $Y_{n-1:0} = \alpha^{i_y}$. Furthermore, there is an isomorphism between the multiplicative group $\{1, \ldots, 2^n\}$ with modulo (2^n+1) multiplication, and the additive group $\{0, \ldots, 2^n-1\}$ with modulo 2^n addition, and:

$$\langle X_{n-1:0} Y_{n-1:0} \rangle_{2^n+1} = g^{\langle i_x + i_y \rangle_{2^n}}.$$

This approach, known as *index calculus*, requires two tables to find the indexes i_x and i_y, and a third table to perform the inverse index operation. Additional hardware is required to handle the special cases where $X_{n-1:0} = 0$ or $Y_{n-1:0} = 0$.

According to [5], look-up-based multipliers are not satisfactory for $n > 8$, and such methods will not be further investigated in this paper.

- A second approach consists in computing the $2n$-bit product $X_{n-1:0}Y_{n-1:0}$ and in performing a subsequent modulo (2^n+1) correction (Section 4.1). This method is for instance well suited to FPGA families embedding multiplier blocks.
- The multiplication of two numbers can be performed in two steps according to the combinational shift-and-add algorithm: i) generation of the partial products $2^i x_i Y_{n-1:0}$; ii) multioperand addition of the partial products. This method can be adapted to modulo (2^n+1) arithmetic: the sum of the modulo-reduced partial products $\langle 2^i x_i Y_{n-1:0} \rangle_{2^n+1}$ is carried out by a multioperand modulo (2^n+1) adder (Section 4.2).
- Finally, a radix-4 modified Booth recoding allows to reduce the number of modulo-reduced partial products by a factor of 2 (Section 4.3).

4.1 Multiplier-Based Operators

Let Φ denote the $2n$-bit product $X_{n-1:0}Y_{n-1:0}$. Since 2^n is congruent with (-1) modulo (2^n+1), we have [1]:

$$\langle \Phi \rangle_{2^n+1} = \langle \Phi_{n-1:0} - \Phi_{2n-1:n} \rangle_{2^n+1}.$$

The modulo (2^n+1) multiplication algorithm defined by (10) is then rewritten as follows:

$$\langle XY \rangle_{2^n+1} = \langle -(\Phi_{2n-1:n} + ((x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0}))) + (\Phi_{n-1:0} + x_n y_n) \rangle_{2^n+1}.$$

If $X = 2^n$ or $Y = 2^n$, the product Φ is equal to zero [21]. Hence,

$$\langle XY \rangle_{2^n+1} = \langle -(\Phi_{2n-1:n} \vee ((x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0}))) + (\Phi_{n-1:0} \vee x_n y_n) \rangle_{2^n+1}. \quad (12)$$

Algorithm 1 summarizes this modulo (2^n+1) multiplication scheme which mainly consists of an unsigned $n \times n$ multiplication and a modular subtraction. Since its operands are n-bit numbers, this last operation is performed according to the algorithm studied in Section 3.1 (Equation (7)).

Algorithm 1 Modulo (2^n+1) multiplication based on an $n \times n$ multiplication.

Require: $0 \leq X, Y \leq 2^n$
Ensure: $P = \langle XY \rangle_{2^n+1}$
1: $\Phi_{2n-1:0} \leftarrow X_{n-1:0} Y_{n-1:0}$;
2: Do in parallel:
  – $A_{n-1:0} \leftarrow \Phi_{n-1:0} \vee x_n y_n$;
  – $B_{n-1:0} \leftarrow \Phi_{2n-1:n} \vee (x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0})$;
3: $P \leftarrow \langle A - B \rangle_{2^n+1}$;

4.2 Adder-Based Operators

Several researchers suggested to rewrite $\langle X_{n-1:0}Y_{n-1:0} \rangle_{2^n+1}$ as a sum of modulo-reduced partial products $\langle 2^i x_i Y_{n-1:0} \rangle_{2^n+1}$, $0 \leq i \leq n-1$. In this Section, we propose an algorithm based on papers by A. Wrzyszcz and D. Milford [21], and R. Zimmermann [23]. We deduce from Property 2 that [23]:

$$\begin{aligned}
\langle X_{n-1:0}Y_{n-1:0} \rangle_{2^n+1} &= \left\langle \sum_{i=0}^{n-1} 2^i x_i Y_{n-1:0} \right\rangle_{2^n+1} \\
&= \left\langle \sum_{i=0}^{n-1} \left(2^i x_i Y_{n-i-1:0} + \overline{x_i Y_{n-1:n-i}} - 2^i + 1\right) \right\rangle_{2^n+1} \\
&= \left\langle n - \underbrace{\sum_{i=0}^{n-1} 2^i}_{=2^n-1} + \sum_{i=0}^{n-1} \Lambda^{(i)} \right\rangle_{2^n+1} \\
&= \left\langle n + 2 + \sum_{i=0}^{n-1} \Lambda^{(i)} \right\rangle_{2^n+1}, \quad (13)
\end{aligned}$$

where

$$\Lambda^{(i)} = 2^i x_i Y_{n-i-1:0} + \overline{x_i Y_{n-1:n-i}}. \quad (14)$$

Substituting the above identity in (11) yields:

$$\langle XY \rangle_{2^n+1} = \left\langle \sum_{i=0}^{n-1} \Lambda^{(i)} + \overline{(x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0})} + x_n y_n + n + 4 \right\rangle_{2^n+1}.$$

By defining $\Lambda^{(n)} = \overline{(x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0})}$ and $\Lambda^{(n+1)} = x_n y_n + 3 = 2^2 x_n y_n + 2\overline{x_n y_n} + \overline{x_n y_n}$, we obtain the following modulo (2^n+1) multiplication scheme:

$$\langle XY \rangle_{2^n+1} = \left\langle \sum_{i=0}^{n+1} \Lambda^{(i)} + n + 1 \right\rangle_{2^n+1}. \quad (15)$$

As all modulo-reduced partial products $\Lambda^{(i)}$ are n-bit numbers, their sum can be carried out by a modulo (2^n+1) diminished-one multioperand adder, which automatically adds the constant $(n+1)$ (see Section 3.2). It is however possible to replace the addition of $\overline{(x_n \oplus y_n)(X_{n-1:0} \vee Y_{n-1:0})}$ by a bitwise OR.

Lemma 2. *Let us define* $\Omega^{(0)} = \Lambda^{(0)} \vee (x_n \vee y_n)(\overline{X_{n-1:0} \vee Y_{n-1:0}})$ *and* $\Omega^{(n)} = 2^2(x_n \vee y_n) + 2(\overline{x_n \vee y_n}) + x_n y_n$. *Then,*

$$\left\langle \Lambda^{(0)} + \Lambda^{(n)} + x_n y_n + 4 \right\rangle_{2^n+1} = \left\langle \Omega^{(0)} + \Omega^{(n)} \right\rangle_{2^n+1}.$$

Suppose now that $\Omega^{(i)} = \Lambda^{(i)}$, $\forall i \in \{1, \ldots, n-1\}$. Modulo (2^n+1) multiplication can be formulated as a sum of $(n+1)$ modulo-reduced partial products and the constant n (Algorithm 2):

$$\langle XY \rangle_{2^n+1} = \left\langle \sum_{i=0}^{n} \Omega^{(i)} + n \right\rangle_{2^n+1}.$$

Algorithm 2 Addition-based modulo (2^n+1) multiplication.

Require: $0 \le X, Y \le 2^n$
Ensure: $P = \langle XY \rangle_{2^n+1}$

1: Compute the n partial products $\Lambda^{(i)}$, $0 \le i \le n-1$, according to (14);
2: Do in parallel:
   – $\Omega^{(0)} \leftarrow \Lambda^{(0)} \vee (x_n \vee y_n)(\overline{X_{n-1:0} \vee Y_{n-1:0}})$;
   – $\Omega^{(i)} \leftarrow \Lambda^{(i)}$, $\forall i \in \{1, \ldots, 2^n-1\}$;
   – $\Omega^{(n)} \leftarrow 2^2(x_n \vee y_n) + 2(\overline{x_n \vee y_n}) + x_n y_n$;
3: Carry out the sum of the $(n+1)$ modulo-reduced partial products $\Omega^{(i)}$ with a modulo (2^n+1) diminished-one multioperand adder:
   $P = \left\langle n + \sum_{i=0}^{n} \Omega^{(i)} \right\rangle_{2^n+1}$;

4.3 Adder-Based Operators with Modified Booth Recoding

Radix-4 modified Booth recoding is a well-known technique to speed-up multiplication. The operand $Y_{n-1:0}$ is for instance recoded into radix-4 digits with values $\{-2, -1, 0, 1, 2\}$, and the product $\langle X_{n-1:0} Y_{n-1:0} \rangle_{2^n+1}$ is rewritten as follows:

$$\langle X_{n-1:0} Y_{n-1:0} \rangle_{2^n+1} = \left\langle \sum_{i=0}^{\lfloor n/2 \rfloor} (-2w_{2i+1} + w_{2i} + w_{2i-1}) 2^{2i} X_{n-1:0} \right\rangle_{2^n+1},$$

where

$$w_i = \begin{cases} y_i & \text{if } i \in \{0, \ldots, n-1\}, \\ 0 & \text{if } i \in \{-1, n, n+1\}. \end{cases}$$

The definition of w_i is essential to distinguish between y_n and the additional bit of weight 2^n involved in the recoding of $Y_{n-1:0}$. When $(-2w_{2i+1} + w_{2i} + w_{2i-1}) \neq 0$, the modulo-reduced partial products are computed according to Property 2 (Table 2). For instance, if $n = 8$, $i = 2$, and $W_{5:3} = (100)_2$, we obtain:

$$\left\langle -2^5 X_{n-1:0} \right\rangle_{2^n+1} = \left\langle \underbrace{2^5 \overline{X_{2:0}} + X_{7:3}}_{\Gamma^{(2)}} + 2 \cdot 2^4 + 1 \right\rangle_{2^n+1}.$$

Thus, each modulo-reduced partial product is the sum of an n-bit number $\Gamma^{(i)}$, a correction term $2^{2i} K_{2i+1:2i}$, where $K_{2i+1:2i} \in \{-2, -1, 1, 2\}$, and the constant 1:

$$\langle X_{n-1:0} Y_{n-1:0} \rangle_{2^n+1} = \left\langle \sum_{i=0}^{\lfloor n/2 \rfloor} \left(\Gamma^{(i)} + 2^{2i} K_{2i+1:2i} + 1 \right) \right\rangle_{2^n+1}.$$

In our example, we have: $\left\langle -2^5 X_{n-1:0} \right\rangle_{2^n+1} = \left\langle \Gamma^{(2)} + 2^4 K_{5:4} + 1 \right\rangle_{2^n+1}$, with $\Gamma^{(2)} = 2^5 \overline{X_{2:0}} + X_{7:3}$ and $K_{5:4} = 2$. The correction term K is a redundant number with digits in $\{-1, 0, 1\}$, which must be converted into a conventional number system prior to its addition. The complexity of this operation depends on the encoding of $\Gamma^{(i)}$ and $K_{2i+1:2i}$ for $W_{2i+1:2i-1} = (000)_2$ and $W_{2i+1:2i-1} = (111)_2$. We adopt the representation proposed by R. Zimmermann in [23], which allows the design of a constant-time conversion algorithm:

$$(\Gamma^{(i)}, K_{2i+1:2i}) = \begin{cases} (2^{2i} - 1, -1) & \text{if } W_{2i+1:2i-1} = (000)_2, \\ (2^n - 2^{2i}, +1) & \text{if } W_{2i+1:2i-1} = (111)_2. \end{cases}$$

Table 2: Modulo-reduced partial products for radix-4 modified Booth recoding [23].

$W_{2i+1:2i-1}$	$\Gamma^{(i)}$	$K_{2i+1:2i}$
000	$2^{2i} - 1$	-1
001 010	$2^{2i} X_{n-2i-1:0} + \overline{X_{n-1:n-2i}}$	-1
011	$2^{2i} X_{n-2i-2:0} + \overline{X_{n-1:n-2i-1}}$	-2
100	$2^{2i} \overline{X_{n-2i-2:0}} + X_{n-1:n-2i-1}$	$+2$
101 110	$2^{2i} \overline{X_{n-2i-1:0}} + X_{n-1:n-2i}$	$+1$
111	$2^n - 2^{2i}$	$+1$

Lemma 3. *Let* $\Gamma^{(\lfloor n/2 \rfloor + 1)}$ *be an* n*-bit unsigned integer such that* K *is congruent with* $\Gamma^{(\lfloor n/2 \rfloor + 1)} + 2$ *modulo* $(2^n + 1)$. *Then,* $\Gamma^{(\lfloor n/2 \rfloor + 1)}$ *can be computed in constant-time by means of Tables 3a, 3b, and 3c.*

R. Zimmermann designed an equivalent algorithm to compute $\Gamma^{(\lfloor n/2 \rfloor + 1)}$ by manipulating logic equations when n is even [23]. It suffices to derive a Karnaugh map from Tables 3b and 3c to obtain the correction term defined by

Table 3: Computation of the correction term $\Gamma^{(\lfloor n/2 \rfloor+1)}$.

$W_{1:0}$ or $\tilde{W}_{1:0}$	$\Gamma^{(\lfloor n/2 \rfloor+1)}_{1:0}$
$(00)_2$	2
$(01)_2$	2
$(10)_2$	1
$(11)_2$	0

(a) n odd

$W_{1:0}$ or $\tilde{W}_{1:0}$	$\Gamma^{(\lfloor n/2 \rfloor+1)}_{1:0}$	
	$w_{n-1}=1$	$w_{n-1}=0$
$(00)_2$	2	3
$(01)_2$	2	3
$(10)_2$	1	2
$(11)_2$	0	1

(b) n even

$W_{2i+1:2i-1}$	$\Gamma^{(\lfloor n/2 \rfloor+1)}_{2i+1:2i}$
$(000)_2$	2
$(001)_2$	3
$(010)_2$	2
$(011)_2$	2
$(100)_2$	1
$(101)_2$	1
$(110)_2$	0
$(111)_2$	1

(c)

$W_{3:1}$	$\Gamma^{(\lfloor n/2 \rfloor+1)}_{3:2}$	
	$x_n=1$	$x_n=0$
$(000)_2$	2	2
$(001)_2$	2	3
$(010)_2$	2	2
$(011)_2$	1	2
$(100)_2$	1	1
$(101)_2$	0	1
$(110)_2$	0	0
$(111)_2$	0	1

(d)

R. Zimmermann. L. A. Sousa studied the modulo (2^n+1) multiplication in the diminished-one number system and also proposed to compute a correction term by rewriting and simplifying logic equations [19]. We believe that the approach introduced in this paper allows a much simpler proof of correctness of such algorithms.

Note that the first modulo-reduced partial product $\Gamma^{(0)}$ is not equal to zero when $X = 2^n$ and $w_1 = 1$. It is therefore impossible to replace the addition of $(x_n \vee y_n)(\overline{X_{n-1:0} \vee Y_{n-1:0}})$ by a bitwise OR, as in Algorithm 2. We solve this problem by defining $\tilde{W}_{1:0} = 2\bar{x}_n y_1 + y_0$: since $\tilde{w}_1 = 0$ when $X = 2^n$, we guarantee that $\Gamma^{(0)} = 0$ and $K_{1:0} = -1$ (Table 2). Consequently, the computation of $\Gamma^{(\lfloor n/2 \rfloor+1)}_{1:0}$ never generates an output carry when $X = 2^n$ and we have to build a new table for $\Gamma^{(\lfloor n/2 \rfloor+1)}_{3:2}$. We deduce from (16) that $\Gamma^{(\lfloor n/2 \rfloor+1)}_{3:2} = 4w_3 - K_{3:2} - 3$, and we obtain Table 3d. We can now perform the modulo (2^n+1) multiplication by adding $\lfloor n/2 \rfloor + 3$ modulo-reduced partial products and the constant $\lfloor n/2 \rfloor + 2$ (Algorithm 3).

4.4 Modified Modulo (2^n+1) Multiplication

The set $(\mathbb{Z}/(2^n+1)\mathbb{Z})^* = \{X \in \mathbb{Z}/(2^n+1)\mathbb{Z} \mid \gcd(X, 2^n+1) = 1\}$ is a group of order $\phi(2^n+1)$ under the operation of multiplication modulo (2^n+1), where ϕ is the *Euler's totient function* (for $m \geq 1$, $\phi(m)$ denotes the number of integers in $\{1, \ldots, m\}$ which are relatively prime to m). Since 2^n belongs to the group, each element is an $(n+1)$-bit number. It is however possible to represent 2^n by 0 ($0 \notin (\mathbb{Z}/(2^n+1)\mathbb{Z})^*$) and to save one bit. This trick implies to define a modified modulo (2^n+1) multiplication operator $\odot$, which replaces an operand equal to 0 by 2^n prior to modular multiplication, and a product equal to 2^n by 0. A modified modulo $(2^{16}+1)$ is for instance the critical arithmetic operation of the IDEA block cipher [11]. Multiplier- and adder-based operators studied in Sections 4 only require a small hardware overhead to perform this operation. Let us convert the inputs $X, Y \in (\mathbb{Z}/(2^n+1)\mathbb{Z})^*$ into radix-2 integers V and W:

$$V = \begin{cases} 2^n & \text{if } X = 0, \\ X & \text{otherwise,} \end{cases} \quad \text{and} \quad W = \begin{cases} 2^n & \text{if } Y = 0, \\ Y & \text{otherwise.} \end{cases}$$

Then, $X \odot Y = \left\langle \langle VW \rangle_{2^n+1} \right\rangle_{2^n}$.

5 Implementation Results and Comparisons

This section describes the implementation of the algorithms discussed in this paper on Xilinx Spartan-IIE and Virtex-II FPGAs. The tables summarize place-and-route results obtained with Xilinx's tools (ISE 5.2.03i).

Our first experiment provides a comparison of modulo (2^n+1) adders (Table 4). Our novel algorithm (Equation (4)) leads to the same area and delay as the basic algorithm defined by (2), while handling the case where $X' = Y' = 2^n$. On Spartan-IIE FPGAs, the area of a multiplexer selecting between two k-bit words is the same as the one of a k-bit CRA. Consequently, the algorithm defined by (3) increases the area by a factor of 1.5 compared to our solution. Finally, parallel-prefix networks do not take advantage of the dedicated carry logic and require a larger area (Sklansky's prefix algorithm was used in this experiment).

Table 5 proposes a comparison of the three multi-operand modulo (2^n+1) addition schemes discussed in Section 3.2:

- The best solution, both in terms of area and delay, consists in computing the sum of the m operands by means of a tree of CRAs, and in performing a modulo correction with a modulo (2^n+1) subtracter (i.e. 2 CRAs). Considering the CRA as the basic building block, we can define the depth of the operator by $d = \lceil \log_2 m \rceil + 2$.

Table 4: Comparison of modulo (2^n+1) adders on an XC2S300E-6 FPGA.

Algorithm	$\sum$	n = 8		n = 16		n = 24		n = 32	
		Area [Slices]	Delay [ns]	Area [Slices]	Delay [ns]	Area [Slices]	Delay [ns]	Area [Slices]	Delay [ns]
Equation (2)	CRA	10	11	18	17	26	18	34	20
Equation (3)	CRA	15	12	27	18	39	19	51	21
Equation (4)	CRA	10	12	18	16	26	18	34	20
	Prefix	39	14	74	19	130	19	144	20

Algorithm 3 Addition-based modulo (2^n+1) multiplication with modified Booth recoding.

Require: $0 \leq X, Y \leq 2^n$
Ensure: $P = \langle XY \rangle_{2^n+1}$
1: $W_{n-1:0} \leftarrow Y_{n-1:0}$; $w_n \leftarrow 0$; $w_{n+1} \leftarrow 0$;
2: $\tilde{W}_{1:0} \leftarrow 2\bar{x}_n y_1 + y_0$;
3: Use $\tilde{W}_{1:0}$ and compute $\Gamma^{(0)}$ according to Table 2;
4: Compute the $\lfloor n/2 \rfloor$ partial products $\Gamma^{(i)}$, $1 \leq i \leq \lfloor n/2 \rfloor$, according to Table 2;
5: **if** n is even **then**
6: Compute $\Gamma^{(\lfloor n/2 \rfloor+1)}_{n-1:2}$ according to Tables 3a and 3b;
7: Use $\tilde{W}_{1:0}$ to compute $\Gamma^{(\lfloor n/2 \rfloor+1)}_{1:0}$ according to Table 3a;
8: **else**
9: Compute $\Gamma^{(\lfloor n/2 \rfloor+1)}_{n-2:2}$ according to Tables 3b, 3c, and 3d;
10: Use $\tilde{W}_{1:0}$ to compute $\Gamma^{(\lfloor n/2 \rfloor+1)}_{1:0}$ according to Table 3b;
11: $\Gamma^{(\lfloor n/2 \rfloor+1)}_{n-1:n-1} \leftarrow \bar{w}_{n-1} w_{n-2}$;
12: **end if**
13: Do in parallel:
– $\Delta^{(0)} \leftarrow \Gamma^{(0)} \vee (x_n \vee y_n)(\overline{X_{n-1:0} \vee Y_{n-1:0}})$;
– $\Delta^{(i)} \leftarrow \Gamma^{(i)}$, $\forall i \in \{1, \ldots, \lfloor n/2 \rfloor + 1\}$;
– $\Delta^{(\lfloor n/2 \rfloor+2)} \leftarrow 2^2(x_n y_n) + 2(x_n \oplus y_n) + \overline{x_n y_n}$;
14: Carry out the sum of the $(\lfloor n/2 \rfloor + 3)$ modulo-reduced partial products $\Delta^{(i)}$ with a modulo (2^n+1) diminished-one multioperand adder:

$$P = \left\langle \lfloor n/2 \rfloor + 2 + \sum_{i=0}^{\lfloor n/2 \rfloor+2} \Delta^{(i)} \right\rangle_{2^n+1};$$

- Our novel modulo (2^n+1) adder requires 2 CRAs (Figure 1). Performing the modulo (2^n+1) multioperand addition by means of a tree of modulo (2^n+1) adders, as suggested in [9], leads to an operator of depth $d = 2\lceil \log_2 m \rceil$, which is therefore larger and slower.

- As mentioned in Section 3.2, CSA-based architectures require 2 LUTs to implement a single FA cell. Thus, the operator based on a CSA tree and a parallel-prefix adder is the more expensive in terms of hardware resources.

To provide a hardware comparison of modulo (2^n+1) multipliers, we implemented the three algorithms described in this paper, and the following methods proposed by other researchers:

Table 5: Multioperand modulo (2^8+1) adders on an XC2S300E-6 FPGA.

Architecture	m = 8		m = 16	
	Area [Slices]	Delay [ns]	Area [Slices]	Delay [ns]
CRA	39	20	76	25
$\langle A' + B' + 1 \rangle_{2^n+1}$	70	28	151	36
CSA & prefix	97	20	208	25

- As mentioned in Section 4.2, our adder-based modulo (2^n+1) multiplier was inspired by R. Zimmermann's work [23]. The main difference between these operators lies in the handling of $X = 2^n$ or $Y = 2^n$. R. Zimmermann implemented (13) by means of a diminished-one modulo (2^n+1) CSA tree which returns two n-bit integers C and S such that $\langle C + S \rangle_{2^n+1} = \langle XY - 1 \rangle_{2^n+1}$. He computed in parallel two n-bit integers $\tilde{C}$ and $\tilde{S}$ to deal with the cases where $X = 2^n$ or $Y = 2^n$ (see Algorithm 4). Then, two multiplexers select the inputs of a final modulo (2^n+1) adder which returns $\langle C + S + 1 \rangle_{2^n+1}$ or $\left\langle \tilde{C} + \tilde{S} + 1 \right\rangle_{2^n+1}$ according to $x_n \vee y_n$.

  Note that the operator described in [23] assumes that the operands belong to $(\mathbb{Z}/(2^n+1)\mathbb{Z})^*$. The end-around-carry parallel-prefix adder returns indeed an n-bit number. Since $XY \neq k \cdot (2^n+1)$, $\forall k \in \mathbb{N}$, the output 0 corresponds to 2^n. However, in the general case, we can not distinguish between 0 and 2^n. For $n = 5$, R. Zimmermann's operator returns for instance $\langle 6 \cdot 11 \rangle_{2^5+1} = 0$ and $\langle 4 \cdot 8 \rangle_{2^5+1} = 0$. Consequently, we implemented this algorithm with the parallel-prefix adder studied in Section 3 for our experiments.

- Y. Ma combined radix-4 modified Booth recoding in the diminished-one number system and Wallace trees [13]. For n even, he demonstrated that

$$\langle XY \rangle_{2^n+1} = \left\langle \sum_{i=0}^{\frac{n}{2}-1} (2^{2i}(-2y'_{2i-1} + y'_{2i} + y'_{2i-1})X - 1) + \frac{n}{2} \right\rangle_{2^n+1},$$

Algorithm 4 R. Zimmermann's modulo (2^n+1) multiplication scheme [23].

Require: $0 \leq X, Y \leq 2^n$
Ensure: $P = \langle XY \rangle_{2^n+1}$
1: Compute the n partial products $\Lambda^{(i)}$, $0 \leq i \leq n-1$, according to (14);
2: Do in parallel:
$- (C, S) \leftarrow \left\langle n+1+\sum_{i=0}^{n-1} \Lambda^{(i)} \right\rangle_{2^n+1}$;
$- (\tilde{C}, \tilde{S}) \leftarrow \begin{cases} (\overline{Y_{n-1:0}}, 1) & \text{if } X = 2^n \text{ and } Y \neq 2^n, \\ (\overline{X_{n-1:0}}, 1) & \text{if } X \neq 2^n \text{ and } Y = 2^n, \\ (0,0) & \text{if } X = 2^n \text{ and } Y = 2^n. \end{cases}$
3: **if** $X \neq 2^n$ and $Y \neq 2^n$ **then**
4: $P \leftarrow \langle C + S + 1 \rangle_{2^n+1}$;
5: **else**
6: $P \leftarrow \left\langle \tilde{C} + \tilde{S} + 1 \right\rangle_{2^n+1}$;
7: **end if**

where $y'_{-1} = \bar{y}'_n \wedge \bar{y}'_{n-1}$. For n odd, he obtained

$$\langle XY \rangle_{2^n+1} = \left\langle (2^{n-1}(y'_{n-1} + y'_{n-2})X - 1) + \sum_{i=0}^{\frac{n-1}{2}-1} (2^{2i}(-2y'_{2i-1} + y'_{2i} + y'_{2i-1})X - 1) + \frac{n-1}{2} + 1 \right\rangle_{2^n+1},$$

where $y'_{-1} = \bar{y}'_n$. Then, he deduced from Property 2 that, if $X \neq 0$, $\langle 2^i X - 1 \rangle_{2^n+1} = \left\langle 2^i X'_{n-i-1:0} + \overline{X'_{n-1:n-i}} \right\rangle_{2^n+1}$ and $\langle -2^i X - 1 \rangle_{2^n+1} = \left\langle 2^i \overline{X'_{n-i-1:0}} + X'_{n-1:n-i} \right\rangle_{2^n+1}$, and got rid of the correction term K introduced in Section 4.3. However, for $X = 0$ (i.e. $X'_n = 1$) or $(-2y'_{2i-1} + y'_{2i} + y'_{2i-1}) = 0$, the resulting modulo-reduced partial product is the n-bit number 2^n. Thus, compared to Algorithm 3, each recoding cell requires the additional bit x'_n (Figure 2). Furthermore, the addition of the modulo-reduced partial products $\Upsilon^{(i)}$ can not be carried out by the CSA-base operator studied in Section 3.2, which supposes that all inputs belong to $\{0, \ldots, 2^n - 1\}$. Y. Ma first computed the sum of the $\Upsilon^{(i)}$'s by means of a CSA and obtained two $(n + \lfloor \log_2 n \rfloor)$-bit numbers for n even, or two $(n + \lfloor \log_2 (n+1) \rfloor)$-bit numbers for n odd. Then, he designed a carry-save correction unit based on Property 1.

Unlike the other algorithms described in this paper, Y. Ma's method requires less hardware resources to implement the modified modulo (2^n+1) multiplication than the classic one [2]. Due to the special encoding of 2^n, the diminished-one representation of a number X is $X' = \langle X - 1 \rangle_{2^n}$. We obtain for example $\langle 0 - 1 \rangle_{2^n} = 2^n - 1$, which is the diminished-one representation of 2^n. Furthermore, X' and Y' now belong to $\{0, \ldots, 2^n - 1\}$, and x'_n is not involved anymore in the computation of the modulo reduced partial products $\Upsilon^{(i)}$.

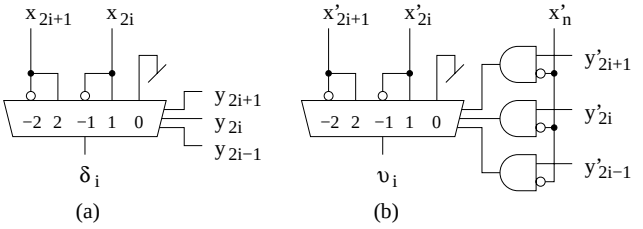


Figure 2: Modulo-reduced partial product generation with modified Booth recoding for (a) Algorithm 3, and (b) Y. Ma's algorithm.

Algorithm 5 Y. Ma's modulo (2^n+1) multiplication scheme [13].

Require: $0 \leq X, Y \leq 2^n$
Ensure: $P = \langle XY \rangle_{2^n+1}$
1: $X' \leftarrow \langle X - 1 \rangle_{2^n+1}$ and $Y' \leftarrow \langle Y - 1 \rangle_{2^n+1}$;
2: **if** n is even **then**
3: $y'_{-1} \leftarrow \bar{y}'_n \wedge \bar{y}'_{n-1}$ and $\Upsilon^{(n/2)} \leftarrow n/2$
4: **else**
5: $y'_{-1} \leftarrow \bar{y}'_n$ and $\Upsilon^{((n-1)/2+1)} \leftarrow (n-1)/2 + 1$;
6: **end if**
7: **for** i in 0 to $\lfloor n/2 \rfloor - 1$ **do**
8: $\Upsilon^{(i)} \leftarrow x'_n 2^n + \bar{x}'_n \cdot \langle 2^{2i}(-2y'_{2i-1} + y'_{2i} + y'_{2i-1})X - 1 \rangle_{2^n+1}$;
9: **end for**
10: **if** n is odd **then**
11: $\Upsilon^{((n-1)/2)} \leftarrow x'_n 2^n + \bar{x}'_n \cdot \langle 2^{n-1}(y'_{n-1} + y'_{n-2})X - 1 \rangle_{2^n+1}$;
12: **end if**
13: $P \leftarrow \left\langle \sum_{i=0}^{\lceil n/2 \rceil} \Upsilon^{(i)} \right\rangle_{2^n+1}$;

- Curiger *et al.* proposed an $\odot$ operator based on an $(n+1) \times (n+1)$ multiplier [5][2]. A. Hiasat independently applied the same idea to implement a classic modulo (2^n+1) multiplication [10] (Algorithm 6). Once he computed the $(2n+1)$-bit product $\Psi = XY$, he had to perform a subsequent modulo (2^n+1) correction defined by:

$$\langle XY \rangle_{2^n+1} = \langle \psi_{2n} 2^{2n} + \Psi_{2n-1:n} 2^n + \Psi_{n-1:0} \rangle_{2^n+1}, = \langle (\Psi_{n-1:0} \vee \psi_{2n}) - \Psi_{2n-1:n} \rangle_{2^n+1}.$$

[2] Note that the operator depicted by Figure 1 in [5] returns for instance $1 \odot 1 = 2$. Replacing the OR gate by a NOT XOR gate corrects the mistake.

Algorithm 6 A. Hiasat's modulo (2^n+1) multiplication scheme [10].

Require: $0 \leq X, Y \leq 2^n$
Ensure: $P = \langle XY \rangle_{2^n+1}$
1: $\Psi_{2n-1:0} \leftarrow XY$;
2: Do in parallel:
   – $A_{n-1:0} \leftarrow \Psi_{n-1:0} \vee \psi_{2n}$;
   – $B_{n-1:0} \leftarrow \Psi_{2n-1:n}$;
3: $P \leftarrow \langle A - B \rangle_{2^n+1}$;

- X. Lai designed an algorithm for software implementations of the modified modulo (2^n+1) multiplication [11] (Algorithm 7). Known as *Low-High algorithm*, this scheme is based on an $n \times n$ multiplication (the cases where $X = 2^n$ or $Y = 2^n$ are handled separately) and a subsequent modulo correction. This last operation is based on the following rewriting of the modulo (2^n+1) subtraction defined by (7):

$$\langle A - B \rangle_{2^n+1} = \begin{cases} (2^n + A) - B + 1 & \text{if } A < B, \\ A - B & \text{otherwise.} \end{cases}$$

  In software, performing the comparison $A < B$ is indeed more efficient than implementing the algorithm defined by (7), where the single computation of d_n involves masking and shifting instructions. However, several hardware implementations of the IDEA block cipher are based on the Low-High algorithm (see for instance [16, 4, 14]), and we will consider it in our comparison.

Algorithm 7 Low-High algorithm [11].

Require: $0 \leq X, Y < 2^n$
Ensure: $P = X \odot Y$
1: **if** $X \neq 0$ and $Y \neq 0$ **then**
2: $\quad \Psi_{2n-1:0} \leftarrow XY$;
3: **else if** $X = 0$ and $Y \neq 0$ **then**
4: $\quad \Psi_{2n-1:0} \leftarrow \langle 2^n + 1 - Y \rangle_{2^n}$;
5: **else**
6: $\quad \Psi_{2n-1:0} \leftarrow \langle 2^n + 1 - X \rangle_{2^n}$;
7: **end if**
8: **if** $\Psi_{2n-1:n} > \Psi_{n-1:0}$ **then**
9: $\quad P \leftarrow \langle \Psi_{n-1:0} - \Psi_{2n-1:n} + 1 \rangle_{2^n}$;
10: **else**
11: $\quad P \leftarrow \langle \Psi_{n-1:0} - \Psi_{2n-1:n} \rangle_{2^n}$;
12: **end if**

Then, we conducted a series of experiments on Xilinx Spartan-IIE and Virtex-II devices. Table 6 indicates that the $n \times n$ multiplication-based algorithm (Algorithm 1) leads to the smaller and faster algorithm on Spartan-IIE FPGAs. This result is related to the architecture of the Spartan-IIE family, which embeds arithmetic logic to improve the efficiency of multiplier implementation; thanks to the MULT_AND gate associated with each LUT, the computation of $x_i y_j + x_r y_s + c_{\text{in}}$ requires a single LUT (Figure 3a). Consider now the sum of the two modulo-reduced partial products $\Lambda^{(2)}$ and $\Lambda^{(3)}$ defined in Section 4.2:

$$\begin{aligned} \Lambda^{(2)} + \Lambda^{(3)} = {} & (x_2 y_{n-3} + x_3 y_{n-4}) 2^{n-1} + \ldots \\ & + (x_2 y_1 + x_3 y_0) 2^3 \\ & + (x_2 y_0 + \overline{x_3 y_{n-1}}) 2^2 \\ & + (\overline{x_2 y_{n-1}} + \overline{x_3 y_{n-2}}) 2^1 \\ & + (\overline{x_2 y_{n-2}} + \overline{x_3 y_{n-3}}). \end{aligned}$$

Synthesis tools still take advantage of the MULT_AND gate to compute $(x_2 y_0 + \overline{x_3 y_{n-1}}) 2^2$. However, the evaluation of $(\overline{x_2 y_{n-1}} + \overline{x_3 y_{n-2}}) 2^1$ or $(\overline{x_2 y_{n-2}} + \overline{x_3 y_{n-3}})$ requires an additional LUT (Figure 3b). The same phenomenon arises for the remaining partial products, thus explaining the results reported in Table 6.

The Spartan-IIE family also provides multiplexers which combine two or four LUTs to respectively compute 5- or 6-input functions. The algorithms based on modified Booth recoding put these features to good use to implement the partial product generators described in Figure 2. Consequently, the generation of $\Delta^{(i)}$ and $\Upsilon^{(i)}$ respectively require $2n$ and $4n$ LUTs. Our experiments show that reducing the number of partial products does not compensate for the cost of such partial product generators on Spartan-IIE FPGAs.

We also performed a series of experiments with CSA-based multioperand adders. We implemented the Sklansky parallel-prefix algorithm to perform the final modulo (2^n+1) addition. Our results indicate that Algorithm 1 and R. Zimmermann's algorithm lead to equivalent operators, both in terms of area and delay. Since we don't have to store $\tilde{C}$, $\tilde{s}_0$, and $x_n y_n$, Algorithm 1 is slightly better if we pipeline the multioperand adder. Besides the removal of the correction unit handling the cases where $X = 2^n$ or $Y = 2^n$, our algorithms have the following advantages in comparison with R. Zimmermann's work:

- Thanks to the modulo (2^n+1) adder described in Section 3, operands are no longer limited to $(\mathbb{Z}/(2^n+1)\mathbb{Z})^*$.
- Tables to perform modified Booth recoding of partial products are provided for any n.

We then studied modified modulo $(2^{16}+1)$ operators for FPGA implementations of the IDEA block cipher. Table 7 indicates that Algorithm 1 is again the best choice for the Spartan-IIE family. The partial products $\Omega^{(i)}$ involved in Algorithm 2 are n-bit numbers. The architecture of the multioperand adder is therefore more regular than the one of the multiplier-based algorithm (Algorithm 1). This explains why pipelining the operator based on Algorithm 1 is more expensive. Finally, these experiments confirm that Y. Ma's algorithm requires less hardware resources for the modified modulo (2^n+1) multiplication than for the classic one.

Virtex-II FPGAs embed multiplier blocks which perform a 17-bit × 17-bit unsigned multiplication or a

Table 6: Comparison of modulo $(2^n + 1)$ multipliers on an XC2S300E-6 FPGA.

Architecture	$\sum$	n = 16		n = 32	
		Area [Slices]	Delay [ns]	Area [Slices]	Delay [ns]
Multiplier-based operator (Algo. 1)	CRA	165	27	655	42
Adder-based operator (Algo. 2)	CRA	266	33	1012	47
	CSA & prefix	504	30	1891	46
Adder-based operator with modified Booth recoding (Algo. 3)	CRA	335	32	1055	43
	CSA & prefix	460	28	1533	35
Zimmermann's operator (Algo. 4)	CSA & prefix	529	30	1829	48
Ma's operator (Algo. 5)	CRA	348	50	1250	70
Hiasat's operator (Algo. 6)	CRA	181	34	694	47

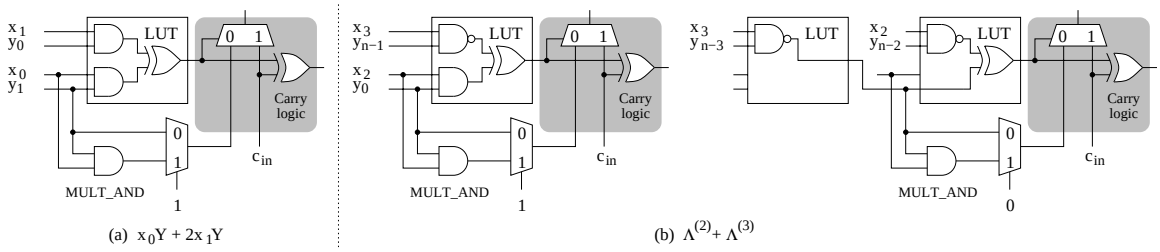


Figure 3: Detail of the computation of partial products on Spartan and Virtex devices.

18-bit × 18-bit two's complement multiplication. For $n = 16$, Algorithm 1 and A. Hiasat's algorithm both require a single block to respectively carry out $\Phi_{2n-1:0} \leftarrow X_{n-1:0}Y_{n-1:0}$ and $\Psi_{2n-1:0} \leftarrow XY$. Consequently, A. Hiasat's algorithm leads to the smallest and fastest operator for the implementation of the IDEA block cipher on Virtex-II devices (Table 8). Due to the comparator and the multiplexer handling the cases where $X = 0$ or $Y = 0$, the $\odot$ operator based on the Low-High algorithm requires about twice as much slices as A. Hiasat's operator.

6 Conclusions

This paper proposed a novel modulo $(2^n + 1)$ addition algorithm suited for both FPGA and ASIC technologies, and described multioperand modulo (2^n+1) adders. Three implementations of a modulo $(2^n + 1)$ multiplication algorithm taking advantage of these adder structures were then discussed and compared with existing solutions.

Our results show that an $n \times n$ multiplication and a subsequent modulo $(2^n + 1)$ correction is the best approach for Spartan-IIE FPGAs. On Virtex-II devices, however, the algorithm proposed by A. Hiasat [10] leads to the smallest and fastest operator. As the results depend on the target technology, our VHDL library is available under GPL license[3], so that researchers and engineers can easily compare the algorithms proposed or described in this paper.

[3] http://perso.ens-lyon.fr/jean-luc.beuchat/ArithLib/

A Proof of Lemma 1

If $X' = Y' = 2^n$, we obtain $x'_n y'_n = x'_n \vee y'_n = 1$, $U_{n-1:0} = 0$, and $\langle X' + Y' + 1 \rangle_{2^n+1} = 2^n$. It remains to prove that our algorithm is equivalent to the one defined by (2) when $X' \neq 2^n$ or $Y' \neq 2^n$. Note that $x'_n y'_n = 0$, $U_{n-1:0} = T_{n-1:0}$, and

$$t_n = \begin{cases} x'_n & \text{if } X' = 2^n \text{ and } Y' \neq 2^n, \\ y'_n & \text{if } X' \neq 2^n \text{ and } Y' = 2^n, \\ u_n & \text{if } X' \neq 2^n \text{ and } Y' \neq 2^n. \end{cases}$$

Consequently, $t_n = x'_n \vee y'_n \vee u_n$ and $U_{n-1:0} + \overline{x'_n \vee y'_n \vee u_n} = T_{n-1:0} + \bar{t}_n$.

B Proof of Lemma 2

Since $\Lambda^{(0)} = 0$ when $(x_n \vee y_n) = 1$ [21], we have

$$\Omega^{(0)} = \begin{cases} \Lambda^{(0)} & \text{if } x_n = y_n = 0, \\ \overline{X_{n-1:0} \vee Y_{n-1:0}} & \text{if } x_n = 1 \text{ or } y_n = 1. \end{cases}$$

For $x_n = y_n = 0$, we obtain $\Lambda^{(n)} = 2^n - 1$, $x_n y_n = 0$, $\Omega^{(n)} = 2$, and

$$\begin{aligned} \left\langle \Lambda^{(0)} + \Lambda^{(n)} + x_n y_n + 4 \right\rangle_{2^n+1} &= \left\langle \Lambda^{(0)} + 2^n + 3 \right\rangle_{2^n+1} \\ &= \left\langle \Lambda^{(0)} + 2 \right\rangle_{2^n+1} \\ &= \left\langle \Omega^{(0)} + \Omega^{(n)} \right\rangle_{2^n+1}. \end{aligned}$$

Table 7: Comparison of modified modulo $(2^{16}+1)$ multipliers on an XC2S300E-6 FPGA.

Architecture	Pipeline	Area [Slices]	Delay [ns]
Multiplier-based operator (Algo. 1)	–	169	30
	3 stages	261	12
Adder-based operator (Algo. 2)	–	277	38
	4 stages	293	12
Ma's operator (Algo. 5)	–	248	37
Hiasat's operator (Algo. 6)	–	186	37
	3 stages	266	14
Low-High (Algo. 7)	–	203	30

Table 8: Comparison of modified modulo $(2^{16}+1)$ multipliers on an XC2V250-5 FPGA.

Architecture	Pipeline	Area [Slices]	Mult. block	Delay [ns]
Multiplier-based operator (Algo. 1)	–	32	✓	15
	3 stages	35	✓	7
Adder-based operator (Algo. 2)	–	224	–	25
	4 stages	233	–	10
Ma's operator (Algo. 5)	–	242	–	28
Hiasat's operator (Algo. 6)	–	25	✓	17
	3 stages	26	✓	8
Low-High (Algo. 7)	–	57	✓	16

For $x_n = y_n = 1$, we have $\Lambda^{(0)} = 0$, $\Lambda^{(n)} = 2^n - 1$, $x_n y_n = 1$, $\Omega^{(0)} = 2^n - 1$, $\Omega^{(n)} = 5$ and

$$\begin{aligned}\left\langle \Lambda^{(0)} + \Lambda^{(n)} + x_n y_n + 4 \right\rangle_{2^n+1} &= \langle 2^n - 1 + 5 \rangle_{2^n+1} \\ &= \left\langle \Omega^{(0)} + \Omega^{(n)} \right\rangle_{2^n+1}.\end{aligned}$$

In the two other cases, $\Lambda^{(0)} = 0$, $\Lambda^{(n)} = \overline{X_{n-1:0} \vee Y_{n-1:0}}$, $x_n y_n = 0$, $\Omega^{(n)} = 2$, and

$$\begin{aligned}&\left\langle \Lambda^{(0)} + \Lambda^{(n)} + x_n y_n + 4 \right\rangle_{2^n+1} \\ &\quad = \langle \overline{X_{n-1:0} \vee Y_{n-1:0}} + 4 \rangle_{2^n+1} \\ &\quad = \left\langle \Omega^{(0)} + \Omega^{(n)} \right\rangle_{2^n+1}.\end{aligned}$$

C Proof of Lemma 3

Remember that $w_n = w_{n+1} = 0$. Thus, the chosen encoding of $K_{2i+1:2i}$ for $W_{2i+1:2i-1} = (000)_2$ guarantees that the correction term K is negative. The basic idea of our conversion algorithm is to compute an n-bit number $\Gamma^{(\lfloor n/2 \rfloor + 1)}$ such that $\langle 2^n + K + 1 \rangle_{2^n+1} = \left\langle \Gamma^{(\lfloor n/2 \rfloor + 1)} + 2 \right\rangle_{2^n+1}$. If n is odd, we have:

$$\langle 2^n + K + 1 \rangle_{2^n+1} = \left\langle 2 + (K_{n:n-1} + 1)2^{n-1} + \sum_{i=0}^{\frac{n-3}{2}} (K_{2i+1:2i} + 3)2^{2i} \right\rangle_{2^n+1}.$$

Since $K_{2i+1:2i}$ belongs to $\{-2, -1, 1, 2\}$ (Table 2), the computation of $K_{2i+1:2i} + 3$ generates a carry bit c_{2i+2}, which must be added to $K_{2i+3:2i+2}$, if and only if $K_{2i+1:2i} > 0$. Consider now the sum $K_{2i+3:2i+2} + 3 + c_{2i+2}$ and note that:

$$K_{2i+3:2i+2} + 3 + c_{2i+2} \in \begin{cases} \{1, 2, 3\} & \text{if } K_{2i+3:2i+2} < 0, \\ \{4, 5, 6\} & \text{if } K_{2i+3:2i+2} > 0. \end{cases}$$

Consequently, the output carry c_{2i+4} does not depend on the input carry c_{2i+2}, and the $K_{2i+3:2i+2} + 3 + c_{2i+2}$ terms can be computed in parallel. Furthermore, we have established that $c_{2i+2} = 1$ when $K_{2i+1:2i} > 0$, i.e. when $w_{2i+1} = 1$ (Table 2). Consequently, $c_{2i+2} = w_{2i+1}$, and $\Gamma^{(\lfloor n/2 \rfloor + 1)}_{2i+1:2i}$ is defined as follows:

$$K_{2i+1:2i} + 3 + w_{2i-1} = 4w_{2i+1} + \Gamma^{(\lfloor n/2 \rfloor + 1)}_{2i+1:2i}. \quad (16)$$

We deduce from Table 2 that

$$K_{2i+1:2i} = \begin{cases} -1 & \text{if } W_{2i+1:2i-1} = (000)_2, \\ +1 & \text{if } W_{2i+1:2i-1} = (111)_2, \\ 2w_{2i+1} - w_{2i} - w_{2i-1} & \text{otherwise.} \end{cases}$$

Therefore

$$\Gamma^{(\lfloor n/2 \rfloor + 1)}_{2i+1:2i} = \begin{cases} 2 & \text{if } W_{2i+1:2i-1} = (000)_2, \\ 1 & \text{if } W_{2i+1:2i-1} = (111)_2, \\ 3 - 2w_{2i+1} - w_{2i} & \text{otherwise.} \end{cases}$$

This equation allows to build Tables 3a and 3c. We apply the same method to compute $K_{n:n-1} + 1 + w_{n-2}$. Since $w_n = 0$, we obtain:

$$\Gamma_{n:n-1}^{(\lfloor n/2 \rfloor + 1)} = \begin{cases} 1 & \text{if } W_{n-1:n-2} = (01)_2, \\ 0 & \text{otherwise,} \end{cases}$$

and the correction term $\Gamma^{(\lfloor n/2 \rfloor + 1)}$ is an n-bit positive integer.

Suppose now that n is even. We have:

$$\begin{aligned} &\langle 2^n + K + 1 \rangle_{2^n+1} \\ &\quad = \left\langle 2^n + 1 + K_{n+1:n} 2^n + \sum_{i=0}^{\frac{n}{2}-1} K_{2i+1:2i} 2^{2i} \right\rangle_{2^n+1}. \end{aligned}$$

Remember that $w_{n+1} = w_n = 0$. Table 2 indicates that $k_{n+1} = 0$ and $k_n = -1$. Thus,

$$\begin{aligned} &\langle 2^n + K + 1 \rangle_{2^n+1} \\ &\quad = \left\langle (2^n + 1) - 2^n + \sum_{i=0}^{\frac{n}{2}-1} K_{2i+1:2i} 2^{2i} \right\rangle_{2^n+1} \\ &\quad = \left\langle (2 + \sum_{i=0}^{\frac{n}{2}-1} 3 \cdot 2^{2i}) - 2^n + \sum_{i=0}^{\frac{n}{2}-1} K_{2i+1:2i} 2^{2i} \right\rangle_{2^n+1} \\ &\quad = \left\langle 2 - 2^n + \sum_{i=0}^{\frac{n}{2}-1} (K_{2i+1:2i} + 3) 2^{2i} \right\rangle_{2^n+1}. \end{aligned}$$

If $w_{n-1} = 1$, we know that the computation of $\Gamma_{n-1:n-2}^{(\lfloor n/2 \rfloor + 1)}$ generates an output carry of weight 2^n canceling the constant -2^n. Otherwise, we perform a modulo $(2^n + 1)$ reduction which simply consists in replacing -2^n by $+1$ to obtain a positive correction term $\Gamma^{(\lfloor n/2 \rfloor + 1)}$:

$$\begin{aligned} \langle -2^n + w_{n-1} 2^n \rangle_{2^n+1} &= \langle -2^n (1 - w_{n-1}) \rangle_{2^n+1} \\ &= \langle 1 - w_{n-1} \rangle_{2^n+1} = \bar{w}_{n-1}. \end{aligned}$$

If $\bar{w}_{n-1} = 0$, we can re-use the results established when n is odd. According to Table 3a, $\Gamma_{1:0}^{(\lfloor n/2 \rfloor + 1)}$ belongs to the set $\{0, 1, 2\}$. Consequently, adding $\bar{w}_{n-1}$ does not generate a carry bit and we can still compute $\Gamma_{2i+1:2i}^{(\lfloor n/2 \rfloor + 1)}$, $i > 0$, by means of Table 3c. A new table allows to compute $\Gamma_{1:0}^{(\lfloor n/2 \rfloor + 1)}$ according to w_{n-1}, w_1, and w_0 (Table 3b).

References

[1] Ramesh C. Agarwal and Charles S. Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-22(2):87–97, April 1974.

[2] Jean-Luc Beuchat. Some modular adders and multipliers for field programmable gate arrays. In *Proceedings of the 17th International Parallel & Distributed Processing Symposium (Reconfigurable Architectures Workshop)*. IEEE Computer Society, 2003.

[3] Jaw John Chang, T. K. Truong, Howard M. Shao, Irving S. Reed, and In-Shek Hsu. The VLSI design of a single chip for the multiplication of integers modulo a Fermat number. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-33(6):1599–1602, December 1985.

[4] O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong. Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA. In C. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, number 2162 in Lecture Notes in Computer Science, pages 333–347. Springer, 2001.

[5] Andreas V. Curiger, Heinz Bonnenberg, and Hubert Kaeslin. Regular VLSI architectures for multiplication modulo $(2^n + 1)$. *IEEE Journal of Solid-State Circuits*, 26(7):990–994, July 1991.

[6] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[7] Dale Gallaher, Frederick E. Petry, and Padmini Srinivasan. The digit parallel method for fast RNS to weighted number system conversion for specific moduli $(2^k - 1, 2^k, 2^k + 1)$. *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 44(1):53–57, January 1997.

[8] Ivan Gonzalez, Sergio Lopez-Buedo, Francisco J. Gómez, and Javier Martinez. Using partial reconfiguration in cryptographic applications: An implementation of the IDEA algorithm. In P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, editors, *Field-Programmable Logic and Applications*, number 2778 in Lecture Notes in Computer Science, pages 194–203. Springer, 2003.

[9] Antti Hämäläinen, Matti Tommiska, and Jorma Skyttä. 6.78 Gigabits per second implementation of the IDEA cryptographic algorithm. In M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications – Reconfigurable Computing Is Going Mainstream*, number 2438 in Lecture Notes in Computer Science, pages 760–769. Springer, 2002.

[10] Ahmad A. Hiasat. New memoryless, mod $(2^n \pm 1)$ residue multiplier. *Electronics Letters*, 28(3):314–315, January 1992.

[11] Xuejia Lai. *On the Design and Security of Block Ciphers*. PhD thesis, Swiss Federal Institute of Technology Zurich, Hartung–Gorre Verlag, 1992.

[12] Lawrence M. Leibowitz. A simplified binary arithmetic for the Fermat number transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-24(5):356–359, October 1976.

[13] Yutai Ma. A simplified architecture for modulo $(2^n + 1)$ multiplication. *IEEE Transactions on Computers*, 47(3):333–337, March 1998.

[14] Allen Michalski, Kris Gaj, and Tarek El-Ghazawi. An implementation comparison of an IDEA encryption cryptosystem on two general-purpose reconfigurable computers. In P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, editors, *Field-Programmable Logic and Applications*, number 2778 in Lecture Notes in Computer Science, pages 204–219. Springer, 2003.

[15] P. V. Ananda Mohan. Comments on "The digit parallel method for fast RNS to weighted number system conversion for specific moduli $(2^k - 1, 2^k, 2^k + 1)$". *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 47(9):972–974, September 2000.

[16] Emeka Mosanya, Christof Teuscher, Hector Fabio Restrepo, Patrick Galley, and Eduardo Sanchez. CryptoBooster: A reconfigurable and modular cryprographic coprocessor. In C. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 1999*, number 1717 in Lecture Notes in Computer Science, pages 246–256. Springer, 1999.

[17] Stanislaw J. Piestrak. A high-speed realization of a residue to binary number system converter. *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 42(10):661–663, October 1995.

[18] Damu Radhakrishnan and Yong Yuan. Novel approaches to the design of VLSI RNS multipliers. *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 39(1):52–57, January 1992.

[19] Leonel A. Sousa. Algorithm for modulo (2^n+1) multiplication. *Electronics Letters*, 39(9):752–754, May 2003.

[20] Frederick J. Taylor. Large moduli multipliers for signal processing. *IEEE Transactions on Circuits and Systems*, CAS-28(7):731–736, July 1981.

[21] Artur Wrzyszcz and David Milford. A new modulo $2^a + 1$ multiplier. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 614–617, 1993.

[22] Reto Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. PhD thesis, Swiss Federal Institute of Technology Zurich, Hartung–Gorre Verlag, 1997.

[23] Reto Zimmermann. Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 158–167. IEEE Computer Society, 1999.

[24] Reto Zimmermann, Andreas V. Curiger, Heinz Bonnenberg, Hubert Kaeslin, Norbert Felber, and Wolfgang Fichtner. A 177 Mbit/s VLSI Implementation of the International Data Encryption Algorithm. *IEEE Journal of Solid-State Circuits*, 29(3):303–307, March 1994.