# Efficient VLSI Implementation of Modulo $(2^n \pm 1)$ Addition and Multiplication

Reto Zimmermann
Swiss Federal Institute of Technology (ETH)
Integrated Systems Laboratory
CH-8092 Zürich, Switzerland
zimmermann@iis.ee.ethz.ch

## Abstract

*New VLSI circuit architectures for addition and multiplication modulo $(2^n - 1)$ and $(2^n + 1)$ are proposed that allow the implementation of highly efficient combinational and pipelined circuits for modular arithmetic. It is shown that the parallel-prefix adder architecture is well suited to realize fast end-around-carry adders used for modulo addition. Existing modulo multiplier architectures are improved for higher speed and regularity. These allow the use of common multiplier speed-up techniques like Wallace-tree addition and Booth recoding, resulting in the fastest known modulo multipliers. Finally, a high-performance modulo multiplier-adder for the IDEA block cipher is presented. The resulting circuits are compared qualitatively and quantitatively, i.e., in a standard-cell technology, with existing solutions and ordinary integer adders and multipliers.*

## 1. Introduction

Arithmetic modulo $(2^n - 1)$ (Mersenne numbers) and modulo $(2^n + 1)$ (Fermat numbers) is used in various applications, e.g., residue number systems (RNS) [11] and cryptography [8]. Efficient and fast modulo adders and multipliers are a prerequisite for corresponding high performance integrated circuits. The main focus in this work is on modulo $(2^n + 1)$ multiplication as used in the IDEA (International Data Encryption Algorithm) block cipher [8]. As tangential results, modulo $(2^n + 1)$ addition and modulo $(2^n - 1)$ addition and multiplication are treated as well. The algorithms for addition are described and compared with existing solutions in Section 2, while the same is done for multiplication in Section 3. Section 4 describes the IDEA modulo multiplier-adder. Experimental results are given in Section 5.

This work has been funded in part by Ascom Systec AG and in part by Microswiss, a Microelectronics Program of the Swiss Government.

### 1.1. Foundations

Binary numbers with $n$ bits are denoted as $A = a_{n-1}a_{n-2}\cdots a_0$ in the following text, where

$$A = \sum_{i=0}^{n-1} 2^i a_i \tag{1}$$

Reduction of a number $A$ modulo a number $M$ ("$A$ mod $M$") can be accomplished by a division (with the remainder as result) or by iteratively subtracting the modulus until $A < M$. For the moduli $(2^n - 1)$ and $(2^n + 1)$, the modulo reduction of a number $A$ with at most $2n$ bits can be computed simply by an addition or subtraction. Since

$$2^n \bmod (2^n - 1) = 2^n - (2^n - 1) = 1 \tag{2}$$

the reduction modulo $(2^n - 1)$ can be formulated as

$$A \bmod (2^n - 1) = (A \bmod 2^n + A \operatorname{div} 2^n) \bmod (2^n - 1) \tag{3}$$

where the modulo operation on the right hand side is used for final correction if the addition yields a result $\geq 2^n - 1$ (i.e., $2^n - 1$ has to be subtracted once). Thus, the modulo $(2^n - 1)$ reduction is computed by adding the high $n$-bit word ($A$ div $2^n$) to the low $n$-bit word ($A$ mod $2^n$) and then conditionally subtracting $2^n - 1$ [5].

Analogously, since

$$2^n \bmod (2^n + 1) = 2^n - (2^n + 1) = -1 \tag{4}$$

the reduction modulo $(2^n + 1)$ can be computed as

$$A \bmod (2^n + 1) = (A \bmod 2^n - A \operatorname{div} 2^n) \bmod (2^n + 1) \tag{5}$$

where the modulo operation on the right hand side is used for final correction if the subtraction yields a negative result (i.e., $2^n + 1$ has to be added once). Thus, the modulo $(2^n + 1)$ reduction is computed by subtracting the high $n$-bit word from the low $n$-bit word and then conditionally adding $2^n + 1$ [5, 13].

Furthermore, the modulo operator has the property that a sum (product) modulo $M$ is equivalent to the sum (product) of its operands modulo $M$:

$$(A+B) \bmod M = (A \bmod M + B \bmod M) \bmod M \quad (6)$$
$$(X \cdot Y) \bmod M = (X \bmod M) \cdot (Y \bmod M) \bmod M \quad (7)$$

## 2. Modulo addition

Modulo carry-propagate addition is the basic operation in modular arithmetic:

$$S = (A+B) \bmod (2^n \pm 1) \quad (8)$$

All known solutions rely on end-around-carry adders and our solution on parallel-prefix adders more particular, both of which are introduced in this section.

### 2.1. Parallel-prefix adders

In a prefix problem, $n$ inputs $x_{n-1}x_{n-2}\cdots x_0$ and an arbitrary associative operator $\bullet$ are used to compute $n$ outputs $y_i = x_i \bullet x_{i-1} \bullet \cdots \bullet x_0$ for $i = 0,\ldots,n-1$. Thus, each output $y_i$ is dependent on all inputs $x_j$ of same or lower magnitude ($j \le i$). Carry propagation in binary addition is a prefix problem [7]. The $n$-bit carry-propagate addition

$$(c_{out}, S) = 2^n c_{out} + S = A + B + c_{in} \quad (9)$$

with input operands $A$ and $B$, carry-in $c_{in}$, sum output $S$, and carry-out $c_{out}$ can be expressed by the logic equations:

*preprocessing:*

$$\begin{aligned} g_i &= \begin{cases} a_0b_0 + a_0c_0 + b_0c_0 & \text{if } i=0 \\ a_ib_i & \text{otherwise} \end{cases} \\ p_i &= a_i \oplus b_i \end{aligned} \quad (10)$$

*prefix computation:*

$$\begin{aligned} (G^0_{i:i}, P^0_{i:i}) &= (g_i, p_i) \\ (G^l_{i:k}, P^l_{i:k}) &= (G^{l-1}_{i:j+1}, P^{l-1}_{i:j+1}) \bullet (G^{l-1}_{j:k}, P^{l-1}_{j:k}) \\ &= (G^{l-1}_{i:j+1} + P^{l-1}_{i:j+1}G^{l-1}_{j:k}, P^{l-1}_{i:j+1}P^{l-1}_{j:k}) \end{aligned} \quad (11)$$

*postprocessing:*

$$\begin{aligned} c_{i+1} &= G^m_{i:0} \\ s_i &= p_i \oplus c_i \end{aligned} \quad (12)$$

for $i = 0,\ldots,n-1$, $l = 1,\ldots,m$, and $0 \le k \le j \le i$ where $a_i$ and $b_i$ are the operand input signals, $g_i$ and $p_i$ the generate and propagate, $c_i$ the carry, and $s_i$ the sum output signals at bit position $i$. $c_0$ and $c_n$ correspond to the carry-in

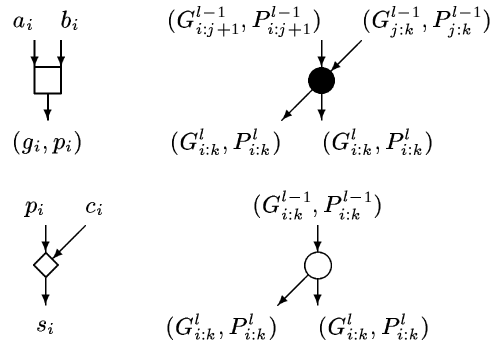

**Figure 1. Prefix adder logic operators.**

$c_{in}$ and carry-out $c_{out}$, respectively. $G^l_{i:k}$ and $P^l_{i:k}$ denote the group generate and propagate signals for the group of bits $i,\ldots,k$ at level $l$. The $\bullet$ operator is repeatedly applied according to a given prefix structure of $m$ levels in order to compute the group generate signal $G^m_{i:0}$ $(= c_{i+1})$ for each bit position $i$.

Prefix structures and adders can be visualized using directed acyclic graphs (DAGs) with the edges standing for signals or signal pairs and the nodes representing the four logic operators depicted in Fig. 1. Fig. 2 shows the general prefix adder structure and Fig. 3 the parallel-prefix structure with the least depth (i.e., resulting in the fastest circuit) [15]. The square (□) and diamond (◇) nodes form the pre- and postprocessing stages, respectively. The black nodes (●) evaluate the prefix operator $\bullet$ and the white nodes (○) pass the signals unchanged to the next prefix level. A variety of other prefix structures with different depths and sizes exist which represent alternative circuit area-delay trade-offs. Also, an efficient algorithm for area optimization of prefix structures under arbitrary depths constraints exists [15].

It is shown in [16] that — at least for cell-based design, e.g., standard cells — the class of prefix adders contains the most efficient adder architectures for the entire range of area-delay trade-offs, i.e., from the smallest ripple-carry adder (serial-prefix) to the fastest carry-lookahead adder (Sklansky parallel-prefix). The simple and highly regular structure of prefix adders allows for easy synthesis, e.g., by netlist generators in pure parameterized VHDL code [17].

### 2.2. End-around-carry adders

In end-around-carry adders, the carry-out is fed back into the carry-in, i.e.,

$$(c_{out}, S) = A + B + c_{out} \quad (13)$$

in order to realize some special function (see below). If done with an ordinary adder, where the carry-out depends on the carry-in, a combinational loop is created that may lead to an unwanted race condition [4]. Different solutions exist:

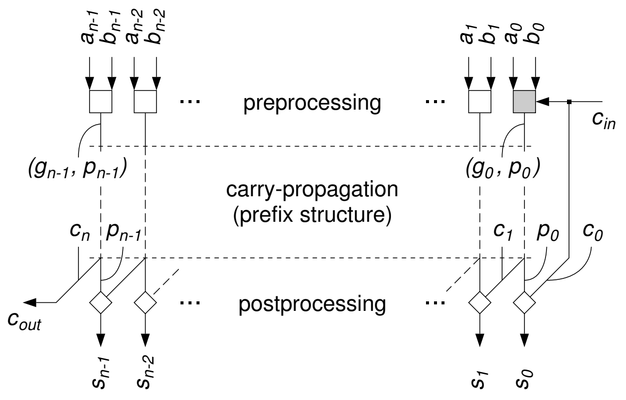

**Figure 2. Prefix adder structure.**

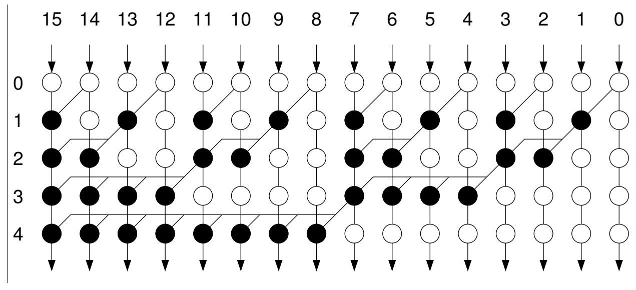

**Figure 3. Parallel-prefix structure by Sklansky.**

a) In some cases, an additional logical operation on the feedback carry can eliminate the race condition [4].

b) The addition is done in two cycles (i.e., the carry-out of the first cycle is added in the second cycle) [2].

c) An adder followed by an incrementer is used.

d) Two adders compute both possible results (i.e., for a carry-in of '0' and '1') in parallel and the correct sum is selected afterwards according to the carry-out.

However, the solutions a)–c) realize two carry propagations in series and thus are slow, while solution d) requires two adders and a multiplexer which results in a large circuit. One approach for fast modulo addition is based on a modification of the traditional carry-lookahead adder [4]. There, the logic formula for the carry-out is re-substituted as carry-in in the logic formulae for the sum bits. Thereby, the carry-lookahead logic is roughly doubled since each sum bit now is a function of all input bits.

In our approach, an adder is required which computes the carry-out independently of the carry-in (i.e., only as carry-out of the sum $A + B$) and which propagates the carry-in to the sum output very quickly (i.e., fast output incrementer) [12].

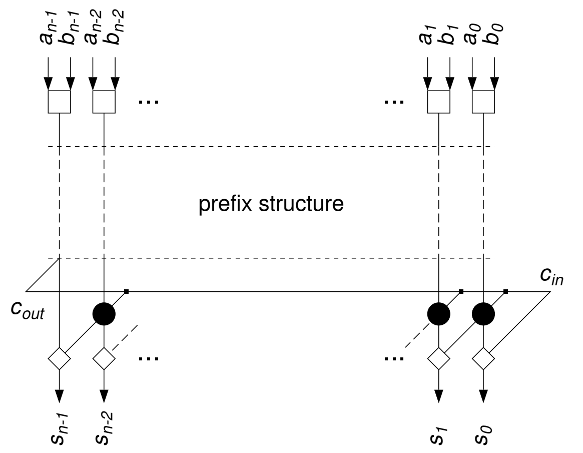
$a_{n-1}$ $b_{n-1}$ $a_{n-2}$ $b_{n-2}$
$a_1$ $b_1$ $a_0$ $b_0$
prefix structure
$c_{in}$
$c_{out}$
$s_{n-1}$ $s_{n-2}$ $s_1$ $s_0$

**Figure 4. End-around-carry parallel-prefix adder structure.**

Since the individual prefix levels in a parallel-prefix adder basically implement incrementer structures (i.e., they compute new generate signals depending on a group-generate input signal of '0' or '1'), an adder with incorporated output incrementer can be built simply by adding an additional prefix level [16]. Fig. 4 depicts the structure of such an end-around-carry parallel-prefix adder. The prefix-structure size is only increased by $n$ black nodes and the critical path by one black node, which results in highly area and delay efficient end-around-carry adders. Note that an $n$-bit end-around-carry parallel-prefix adder has the same delay but is smaller compared to an ordinary $2n$-bit parallel-prefix adder.

## 2.3. Modulo $(2^n - 1)$ addition

Modulo $(2^n - 1)$ addition or, which is the same, one's complement addition can be formulated as

$$(A+B) \bmod (2^n-1) = \begin{cases} A+B-(2^n-1) \\ \quad = (A+B+1) \bmod 2^n \\ \qquad \text{if } A+B \geq 2^n-1 \\ A+B \quad \text{otherwise} \end{cases} \tag{14}$$

The modulo $2^n$ reduction is automatically performed if an $n$-bit adder is used. Note that the value "$11\cdots1$" never occurs and that only one single representation "$00\cdots0$" of zero exists. Equation (14) can be rewritten using the condition $A + B \geq 2^n$:

$$(A+B) \bmod (2^n-1) = \begin{cases} A+B-(2^n-1) \\ \quad = (A+B+1) \bmod 2^n \\ \qquad \text{if } A+B \geq 2^n \\ A+B \quad \text{otherwise} \end{cases} \tag{15}$$

Now, zero has a double representation ("$00 \cdots 0$" and "$11 \cdots 1$"). Since the new condition $A + B \geq 2^n$ is equivalent to $c_{out} = 1$, where $c_{out}$ is the carry-out of the addition $A + B$, equation (15) can be rewritten as

$$(A + B) \bmod (2^n - 1) = (A + B + c_{out}) \bmod 2^n \quad (16)$$

which basically is equivalent to (13). Therefore, modulo $(2^n - 1)$ addition with a double representation of zero can be realized by the $n$-bit end-around-carry parallel-prefix adder of Fig. 4 with $c_{in} = c_{out}$.

The additional condition of $A + B = 2^n - 1 = 11 \cdots 1$ found in (14) is equivalent to $P^m_{n-1:0} = 1$ (i.e., group propagate signal computed in a prefix adder). Therefore, modulo $(2^n - 1)$ addition with a single representation of zero can be realized by the end-around-carry parallel-prefix adder with $c_{in} = c_{out} + P^m_{n-1:0}$ (i.e., with an additional OR-gate in the carry-feedback path).

## 2.4. Modulo $(2^n + 1)$ addition

**Diminished-one number representation.** For modulo $(2^n + 1)$ addition, the diminished-one number system is often used, where the number $A$ is represented by $A' = A - 1$ and the value 0 is not used or treated separately [1] (i.e., requires an additional zero-indication bit which is omitted here). Ordinary addition in this number system looks as follows:

$$\begin{aligned} A + B &= S \\ (A' + 1) + (B' + 1) &= S' + 1 \\ A' + B' + 1 &= S' \end{aligned} \quad (17)$$

Modulo $(2^n + 1)$ addition can now be formulated as

$$(A' + B' + 1) \bmod (2^n + 1) = \begin{cases} A' + B' + 1 - (2^n + 1) \\ \quad = (A' + B') \bmod 2^n \\ \qquad \text{if } A' + B' \geq 2^n \\ A' + B' + 1 \quad \text{otherwise} \end{cases} \quad (18)$$

The sum $A' + B'$ is incremented if $A' + B' < 2^n$, i.e., if $c_{out} = 0$. Thus, modulo $(2^n + 1)$ addition can be realized by the end-around-carry parallel-prefix adder with $c_{in} = \overline{c}_{out}$ (i.e., with an inverter in the carry-feedback path):

$$(A' + B' + 1) \bmod (2^n + 1) = (A' + B' + \overline{c}_{out}) \bmod 2^n \quad (19)$$

The diminished-one number representation, however, often requires the conversion from and to the normal number representation using incrementation/decrementation, which might be too expensive when compared to its advantages.

**Normal number representation.** Equation (19) can also be used for the modulo $(2^n + 1)$ addition of numbers in normal representation

$$(A + B + 1) \bmod (2^n + 1) = (A + B + \overline{c}_{out}) \bmod 2^n \quad (20)$$

with the property that $S + 1$ is computed (i.e., an extra '1' is added). In many applications, such as multipliers (see Section 3), this property can easily be dealt with. Here, the value $2^n$ must be treated separately as a special case.

## 2.5. Modulo carry-save addition

A carry-save adder adds three $n$-bit input operands $A_1$, $A_2$, and $A_3$ without carry-propagation, yielding a redundant sum represented by a sum-bit vector $S = s_{n-1} s_{n-2} \cdots s_0$ and a carry-bit vector $C = c_n c_{n-1} \cdots c_1$:

$$(C, S) = 2C + S = A_1 + A_2 + A_3 \quad (21)$$

It is composed of $n$ full-adders arranged in parallel and has constant delay [6, 14]. $m - 2$ carry-save adders can be arranged in a linear or tree structure for fast addition of $m$ operands, resulting in an adder array or adder tree (Wallace tree), respectively [6]. In an adder array, the carry-save adder at level $l$ with redundant sum output $(C^l, S^l)$ adds the addition operand $A_l$ to the redundant sum output $(C^{l-1}, S^{l-1})$ of the adder at level $l - 1$:

$$(C^l, S^l) = A_l + S^{l-1} + c^{l-1}_{n-1} \cdots c^{l-1}_1 c^l_{in} \quad (22)$$

where $c^l_{in}$ can be regarded as carry-in and $c^l_{out} = c^l_n$ as carry-out of the carry-save adder. Analogously to the end-around carry-propagate adders in the previous text, a modulo $(2^n - 1)$ end-around carry-save adder array can be realized by feeding the carry-out $c^{l-1}_{out}$ back into the carry-in $c^l_{in}$ of the next level, i.e., $c^l_{in} = c^{l-1}_{out}$. A modulo $(2^n + 1)$ carry-save adder array is realized by inverting the feedback carry, i.e., $c^l_{in} = \overline{c}^{l-1}_{out}$. The same principle of feeding back the carry-outs into the carry-ins can also be applied to adder trees. Fig. 5 shows an $m$-operand end-around carry-save adder using (m,2)-compressors ($a_{l,i}$ denotes the $i$-th bit of $A_l$). As an example, Fig. 6 gives an (8,2)-compressor with linear structure for adder arrays (slower, more regular) and with tree structure for adder trees (faster, less regular).

Multi-operand adders can now be built using a modulo carry-save adder array or tree and a final modulo carry-propagate adder. The resulting circuits are very similar to ordinary multi-operand adders but more regular, since the carry-outs have not to be accumulated but can be fed back into the adder structure as carry-ins. Note that modulo $(2^n + 1)$ adders with normal number representation require an additional correction term due to the property of (20).

## 2.6. Discussion

The proposed modulo carry-propagate adders are superior to the solutions based on two carry propagations from the literature [2]. It is also assumed that they result in smaller circuits than the modified carry-lookahead adder from [4]. A quantitative comparison with the latter, however, has not been carried out due to its complex circuit structure.

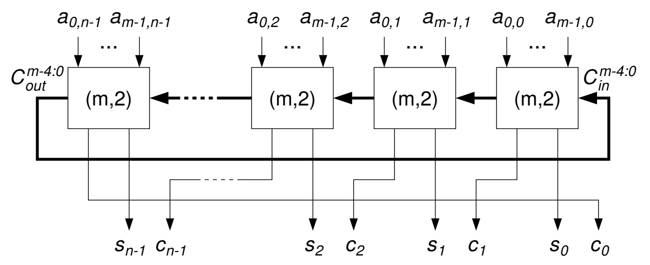

**Figure 5. m-operand end-around carry-save adder using (m,2)-compressors.**

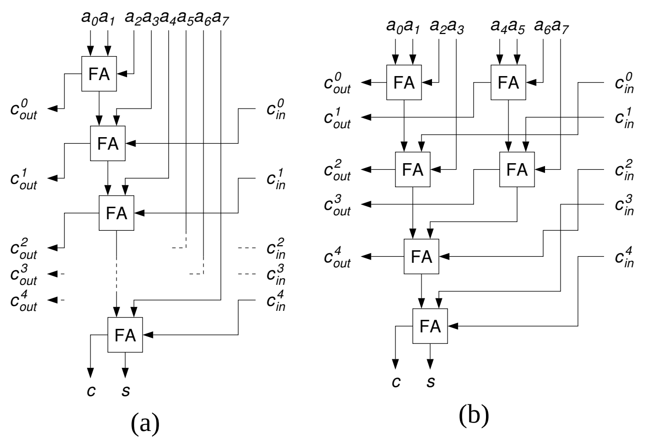
$a_0a_1$ $a_2a_3a_4a_5a_6a_7$
FA
$c_{out}^0$ $c_{out}^1$ $c_{out}^2$ $c_{out}^3$ $c_{out}^4$
$c_{in}^0$ $c_{in}^1$ $c_{in}^2$ $c_{in}^3$ $c_{in}^4$
c s
(a)
$a_0a_1$ $a_2a_3$ $a_4a_5$ $a_6a_7$
(b)

**Figure 6. (a) Linear- and (b) tree-structured (8,2)-compressor.**

## 3. Modulo multiplication

For modulo multiplication,

$$P = X \cdot Y \bmod (2^n \pm 1) \tag{23}$$

various ROM-based solutions using table-lookup have been proposed and compared [10, 3]. Sophisticated methods exist to reduce the table sizes by combining smaller table-lookups with simple arithmetic operations, such as additions. For word lengths larger than eight bits, however, these solutions still require prohibitively large ROMs or many clock cycles for evaluation.

For high-performance modulo multiplication, dedicated multipliers are required which can be implemented as combinational or pipelined circuits. Solutions based on ordinary integer multiplication with subsequent modulo correction using adders are proposed in [3, 5]. A modulo $(2^n+1)$ multiplier architecture with modulo-reduced, Booth-recoded partial products and with concurrent modulo reduction during carry-save addition is proposed in [3] and improved in [9]. It is shown in [13] that modulo $(2^n + 1)$ multipliers with highly regular modulo carry-save adder arrays and trees can be realized.

In this paper, the multiplier from [13], which bases on the diminished-one number representation, is improved by eliminating the precomputation of a correction term $Z$ (i.e., counts the number of '0' in the multiplier $X$) and by using a faster final adder. Also, the algorithm is extended for Booth recoding and for modulo $(2^n - 1)$ multiplication as well as for modulo $(2^n + 1)$ multiplication using normal number representation.

### 3.1. Modulo $(2^n - 1)$ multiplication

According to (3), modulo $(2^n - 1)$ multiplication can be formulated as

$$\begin{aligned} &X \cdot Y \bmod (2^n - 1) \\ &\quad = (X \cdot Y \bmod 2^n + X \cdot Y \operatorname{div} 2^n) \bmod (2^n - 1) \end{aligned} \tag{24}$$

where $X \cdot Y \bmod 2^n$ corresponds to the low output word and $X \cdot Y \operatorname{div} 2^n$ to the high output word of the multiplication $X \cdot Y$. Therefore, modulo $(2^n - 1)$ multiplication can be accomplished by an $n$-bit unsigned multiplication followed by an $n$-bit modulo $(2^n - 1)$ addition. The major drawback of this solution is that two carry-propagate adders in series are required (i.e., one as final adder in the multiplier and one in the modulo adder), resulting in a larger and considerably slower circuit compared to an ordinary multiplier. On the other hand, a standard unsigned multiplier can be used for modulo multiplication.

However, one carry-propagate addition can be saved if the redundant product $(P_C, P_S)$ after the carry-save adder (i.e., before the final adder) is already reduced by the modulus. Then, the addition of (24) is not required anymore and one single modulo $(2^n - 1)$ adder is sufficient to resolve the redundant product representation. A modulo-reduced redundant product $(P_C, P_S)$ can be obtained by

1. modulo-reducing the partial products [3], and
2. using modulo carry-save addition to add them up.

Equation (24) can be rewritten as sum of partial products:

$$\begin{aligned} &X \cdot Y \bmod (2^n - 1) \\ &\quad = \sum_{i=0}^{n-1} 2^i x_i \cdot Y \bmod (2^n - 1) \\ &\quad = \sum_{i=0}^{n-1} x_i \cdot \left(2^i Y \bmod (2^n - 1)\right) \bmod (2^n - 1) \\ &\quad = \sum_{i=0}^{n-1} x_i \cdot (2^i Y \bmod 2^n + 2^i Y \operatorname{div} 2^n) \bmod (2^n - 1) \end{aligned} \tag{25}$$

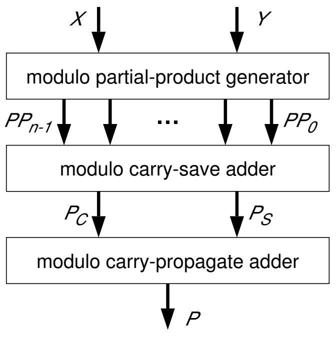
X
Y
modulo partial-product generator
PPn-1
...
PP0
modulo carry-save adder
PC
PS
modulo carry-propagate adder
P

**Figure 7. Modulo $(2^n - 1)$ multiplier architecture.**

$$
\begin{aligned}
&= \sum_{i=0}^{n-1} x_i \cdot (y_{n-i-1} \cdots y_0 0 \cdots 0 \; + \; 0 \cdots 0 y_{n-1} \cdots y_{n-i}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \bmod (2^n - 1) \\
&= \sum_{i=0}^{n-1} x_i \cdot y_{n-i-1} \cdots y_0 y_{n-1} \cdots y_{n-i} \bmod (2^n - 1) \\
&= \sum_{i=0}^{n-1} PP_i \bmod (2^n - 1)
\end{aligned}
$$

where $PP_i = x_i \cdot y_{n-i-1} \cdots y_0 y_{n-1} \cdots y_{n-i}$ (implemented using AND-gates) is the $i$-th partial product modulo $(2^n - 1)$. Note that all $n$-bit partial products $PP_i$ have the same magnitude (as opposed to ordinary multiplication, where the partial products have increasing magnitude), i.e., the number of product bits to add is the same for all bit positions. This allows their addition by a highly regular modulo carry-save adder composed of $n$ ($n$,2)-compressors, yielding the modulo-reduced redundant product $(P_C, P_S)$. Fig. 7 depicts the multiplier architecture with the partial-product generation, $n$-operand carry-save addition, and carry-propagate addition steps, which are all performed modulo $(2^n - 1)$ (note that all signal buses are $n$ bits wide).

**Wallace-tree addition.** The first speed-up technique for multiplication is to accelerate the addition of the partial products using a carry-save adder tree (Wallace tree) [6]. This technique is easily applicable to modulo carry-save adders (and thus to modulo multipliers), as already described in Section 2. The resulting tree structures are even more regular than in ordinary multipliers, because the same number of bits is added for each bit position and the carry-outs are fed back into the carry-ins. In cell-based design, the lower regularity of tree structures compared to linear ones has a negligible impact on circuit area, while a considerable speed-up is achieved. Therefore, the use of carry-save adder trees is always recommended.

**Booth recoding.** The second speed-up technique for multiplication is to reduce the number of partial products by applying bit-pair recoding (Booth recoding) [6]. Equation (1) can be rewritten for the multiplier $X$ as

**Table 1. Bit-pair recoded modulo $(2^n - 1)$ partial products.**

| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | $+\ 0$ | $0 \cdots 0 0 \cdots 0$ |
| 0 | 0 | 1 | $+\ Y$ | $y_{n-2i-1} \cdots y_0 \, y_{n-1} \cdots y_{n-2i}$ |
| 0 | 1 | 0 | $+\ Y$ | $y_{n-2i-1} \cdots y_0 \, y_{n-1} \cdots y_{n-2i}$ |
| 0 | 1 | 1 | $+2Y$ | $y_{n-2i-2} \cdots y_0 \, y_{n-1} \cdots y_{n-2i-1}$ |
| 1 | 0 | 0 | $-2Y$ | $\overline{y}_{n-2i-2} \cdots \overline{y}_0 \, \overline{y}_{n-1} \cdots \overline{y}_{n-2i-1}$ |
| 1 | 0 | 1 | $-\ Y$ | $\overline{y}_{n-2i-1} \cdots \overline{y}_0 \, \overline{y}_{n-1} \cdots \overline{y}_{n-2i}$ |
| 1 | 1 | 0 | $-\ Y$ | $\overline{y}_{n-2i-1} \cdots \overline{y}_0 \, \overline{y}_{n-1} \cdots \overline{y}_{n-2i}$ |
| 1 | 1 | 1 | $-\ 0$ | $0 \cdots 0 0 \cdots 0$ |

$$X = \sum_{i=0}^{n/2} 2^{2i} \underbrace{(x_{2i-1} + x_{2i} - 2x_{2i+1})}_{\{-2,-1,0,+1,+2\}} \tag{26}$$

where $x_{n+1}, x_n, x_{-1} = 0$. The resulting $n/2 + 1$ bit pairs $(x_{2i+1}, x_{2i})$ are used to specify $n/2 + 1$ partial products according to Table 1 (note that the third bit $x_{2i-1}$ must also be considered), which are summed up as follows:

$$X \cdot Y \bmod (2^n - 1) = \sum_{i=0}^{n/2} PP_i \bmod (2^n - 1) \tag{27}$$

The carry-save adder is thereby cut in half (i.e., only half the number of partial products have to be added) while some recoding logic is added. With respect to circuit delay, the recoding logic is roughly compensated by the shallower adder tree (note that in an adder tree, only about two full-adders are saved on the critical path if the number of operands is cut in half). Delay can only be reduced if a carry-save adder array is used. With respect to circuit area, it has been observed that — at least for cell-based design using efficient full-adder cells — the additional recoding logic is not necessarily compensated by the smaller carry-save adder. Therefore, bit-pair recoding not always yields faster and smaller multiplier circuits (see the results in Section 5).

### 3.2. Modulo $(2^n + 1)$ multiplication

Modulo $(2^n + 1)$ multiplication is considered here for application in the IDEA cipher. That is, $n$-bit numbers in normal representation are used for operands and result, where the value 0 is not used and the value $2^n$ is represented by "$00 \cdots 0$". The presented algorithm can easily be adapted for number representations with the value 0 included and the value $2^n$ indicated by a separate bit.

According to (5), modulo $(2^n+1)$ multiplication using the normal number representation can be formulated as

$$\begin{aligned} & X \cdot Y \bmod (2^n+1) && (28) \\ &= \quad (X \cdot Y \bmod 2^n - X \cdot Y \operatorname{div} 2^n) \bmod (2^n+1) \end{aligned}$$

Likewise to modulo $(2^n-1)$ multiplication, an $n$-bit unsigned multiplication followed by an $n$-bit modulo $(2^n+1)$ subtraction can be performed [3]. Again, the multiplication can be accelerated by performing partial-product generation and carry-save addition modulo $(2^n+1)$.

Equation (28) can be rewritten as sum of partial products:

$$\begin{aligned} & X \cdot Y \bmod (2^n+1) && (29) \\ &= \sum_{i=0}^{n-1} 2^i x_i \cdot Y \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} x_i \cdot \big(2^i Y \bmod (2^n+1)\big) \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} x_i \cdot (2^i Y \bmod 2^n - 2^i Y \operatorname{div} 2^n) \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} x_i \cdot (y_{n-i-1} \cdots y_0 0 \cdots 0 - \\ &\qquad 0 \cdots 0 y_{n-1} \cdots y_{n-i}) \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} \big(x_i \cdot (y_{n-i-1} \cdots y_0 0 \cdots 0 - \\ &\qquad 0 \cdots 0 y_{n-1} \cdots y_{n-i}) + \\ &\qquad \overline{x}_i \cdot (-0 \cdots 00 \cdots 0)\big) \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} \big(x_i \cdot (y_{n-i-1} \cdots y_0 0 \cdots 0 + \\ &\qquad 1 \cdots 1 \overline{y}_{n-1} \cdots \overline{y}_{n-i} + 2) + \\ &\qquad \overline{x}_i \cdot (1 \cdots 11 \cdots 1 + 2)\big) \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} \big(x_i \cdot (y_{n-i-1} \cdots y_0 0 \cdots 0 + \\ &\qquad 0 \cdots 0 \overline{y}_{n-1} \cdots \overline{y}_{n-i}) + \\ &\qquad \overline{x}_i \cdot 0 \cdots 01 \cdots 1 + \\ &\qquad (x_i + \overline{x}_i) \cdot (1 \cdots 10 \cdots 0 + 2)\big) \bmod (2^n+1) \\ &= \sum_{i=0}^{n-1} \big(x_i \cdot y_{n-i-1} \cdots y_0 \overline{y}_{n-1} \cdots \overline{y}_{n-i} + \\ &\qquad \overline{x}_i \cdot 0 \cdots 01 \cdots 1 + 1 + \\ &\qquad 1 \cdots 10 \cdots 0 + 1\big) \bmod (2^n+1) \\ &= \Big(\sum_{i=0}^{n-1} (x_i \cdot y_{n-i-1} \cdots y_0 \overline{y}_{n-1} \cdots \overline{y}_{n-i} + \\ &\qquad \overline{x}_i \cdot 0 \cdots 01 \cdots 1 + 1) \\ &\qquad + 2\Big) \bmod (2^n+1) \\ &= \Big(\sum_{i=0}^{n-1} (PP_i + 1) + 2\Big) \bmod (2^n+1) \end{aligned}$$

where "$0 \cdots 01 \cdots 1$" denotes the number with $n-i$ '0' and $i$ '1'. The complement modulo $(2^n+1)$ is computed as

$$(-A) \bmod (2^n+1) = \overline{A} + 2 \bmod (2^n+1) \qquad (30)$$

The term $\overline{x}_i \cdot (-0 \cdots 00 \cdots 0) = \overline{x}_i \cdot (1 \cdots 11 \cdots 1 + 2)$ is added in (29) so that the constant $1 \cdots 10 \cdots 0 + 2$ can be factored out in order to get simpler partial products. Also, the data-dependent correction term $Z$ used in [13] can be eliminated this way. A '1' is added to each partial product in the second last equation of (29) for their modulo $(2^n+1)$ addition, as required in (20). The sum of the remaining constants can be represented by one single constant term:

$$\sum_{i=0}^{n-1} (\underbrace{1 \cdots 1}_{(n-i)\times} \underbrace{0 \cdots 0}_{i\times} + 1) \bmod (2^n+1) = 2 \qquad (31)$$

Thus, modulo $(2^n+1)$ multiplication is performed by adding the modulo-reduced partial products $PP_i = x_i \cdot y_{n-i-1} \cdots y_0 \overline{y}_{n-1} \cdots \overline{y}_{n-i} + \overline{x}_i \cdot 0 \cdots 01 \cdots 1$ (implemented using simplified multiplexers due to constant inputs) and the constant 2 by an $(n+1)$-operand carry-save addition and a final carry-propagate addition, which are all performed modulo $(2^n+1)$. Note that a total of $n$ modulo $(2^n+1)$ additions are carried out which, according to (20), also add the $n$ '1' found in the last equation of (29).

The value $2^n$, which in our case is represented by 0 (and otherwise by an extra bit), must be treated separately. The following cases have to be distinguished:

$$P = \begin{cases} 2^n \cdot Y \bmod (2^n+1) = (-Y) \bmod (2^n+1) & \\ \quad = (\overline{Y} + 2) \bmod (2^n+1) & \text{if } X = 2^n \\ 2^n \cdot X \bmod (2^n+1) = (-X) \bmod (2^n+1) & \\ \quad = (\overline{X} + 2) \bmod (2^n+1) & \text{if } Y = 2^n \\ 2^n \cdot 2^n \bmod (2^n+1) = 1 & \text{if } X = Y = 2^n \\ X \cdot Y \bmod (2^n+1) & \text{otherwise} \end{cases} \qquad (32)$$

A $2^n$-correction unit is required to compute the redundant product

$$(P'_C, P'_S) = \begin{cases} (\overline{Y}, 1) & \text{if } X = 2^n \\ (\overline{X}, 1) & \text{if } Y = 2^n \\ (0, 0) & \text{if } X = Y = 2^n \end{cases} \qquad (33)$$

which is then selected by a multiplexer before the final adder. Note that the constants from (32) are diminished by 1 in (33) because the final modulo adder adds an extra '1'. With $2^n$ represented by 0, the correction unit requires two zero-detectors which, however, are not on the critical path. One additional multiplexer is on the critical path through the multiplier. Fig. 8 depicts the architecture of the modulo $(2^n+1)$ multiplier.

**Wallace-tree addition.** As in modulo $(2^n-1)$ multiplication, adder trees can be applied very easily to speed up carry-save addition in modulo $(2^n+1)$ multiplication.

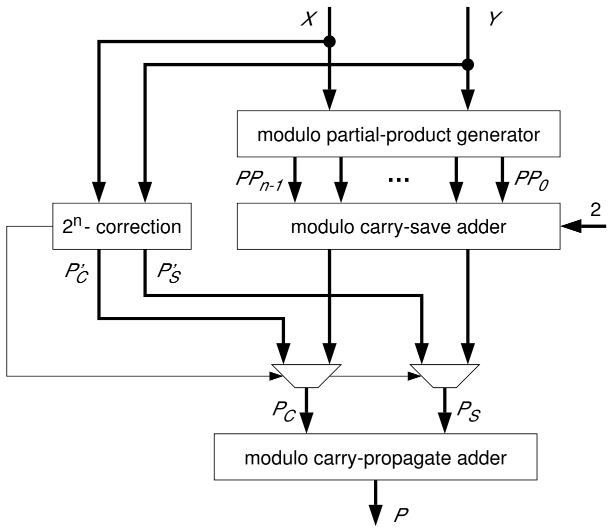

**Figure 8. Modulo $(2^n + 1)$ multiplier architecture.**

**Booth recoding.** Bit-pair recoding to reduce the number of partial products is also possible for modulo $(2^n + 1)$ multiplication. An additional correction term $T$ is required which depends on the multiplier $X$ by the logic equations:

$$\begin{aligned} t_0 &= \overline{x}_1\overline{x}_0\overline{x}_{n-1} + x_1\overline{x}_0 x_{n-1} + x_0\overline{x}_{n-1} \\ t_1 &= \overline{x}_1 + \overline{x}_0\overline{x}_{n-1} \\ t_{2i} &= x_{2i+1}\overline{x}_{2i} + x_{2i+1}x_{2i-1} + \overline{x}_{2i}x_{2i-1} \\ t_{2i+1} &= \overline{x}_{2i+1} \end{aligned} \tag{34}$$

for $i = 1, \ldots, n/2 - 1$. Also, the additional constant is 1 instead of 2. The derivation of the constant and correction terms is not given here due to its complexity. The terms have been exhaustively verified in a circuit implementation.

The $n/2 + 1$ partial products are given in Table 2 and summed up as follows:

$$\begin{aligned} & X \cdot Y \bmod (2^n + 1) \qquad (35) \\ &= \Big( \sum_{i=0}^{n/2} (PP_i + 1) + 1 + T \Big) \bmod (2^n + 1) \end{aligned}$$

## 3.3. Diminished-one multiplication

The modulo $(2^n + 1)$ multiplication algorithm of Fig. 8 can easily be adapted for the diminished-one number representation of input operands and output product [13]:

$$\begin{aligned} P &= X \cdot Y \bmod (2^n + 1) \\ P' + 1 &= (X' + 1) \cdot (Y' + 1) \bmod (2^n + 1) \\ P' + 1 &= (X' \cdot Y' + X' + Y' + 1) \bmod (2^n + 1) \\ P' &= (X' \cdot Y' + X' + Y') \bmod (2^n + 1) \end{aligned} \tag{36}$$

**Table 2. Bit-pair recoded modulo $(2^n + 1)$ partial products.**

| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | $+\ \ 0$ | $0\cdots 0\,1\cdots 1$ |
| 0 | 0 | 1 | $+\ \ Y$ | $y_{n-2i-1}\cdots y_0\,\overline{y}_{n-1}\cdots\overline{y}_{n-2i}$ |
| 0 | 1 | 0 | $+\ \ Y$ | $y_{n-2i-1}\cdots y_0\,\overline{y}_{n-1}\cdots\overline{y}_{n-2i}$ |
| 0 | 1 | 1 | $+\,2Y$ | $y_{n-2i-2}\cdots y_0\,\overline{y}_{n-1}\cdots\overline{y}_{n-2i-1}$ |
| 1 | 0 | 0 | $-\,2Y$ | $\overline{y}_{n-2i-2}\cdots\overline{y}_0\,y_{n-1}\cdots y_{n-2i-1}$ |
| 1 | 0 | 1 | $-\ \ Y$ | $\overline{y}_{n-2i-1}\cdots\overline{y}_0\,y_{n-1}\cdots y_{n-2i}$ |
| 1 | 1 | 0 | $-\ \ Y$ | $\overline{y}_{n-2i-1}\cdots\overline{y}_0\,y_{n-1}\cdots y_{n-2i}$ |
| 1 | 1 | 1 | $-\ \ 0$ | $1\cdots 1\,0\cdots 0$ |

Thereby, the two additional terms $X'$ and $Y'$ have to be added in the modulo carry-save adder, resulting in only a small area and delay increase. The special case of $X, Y = 0$ has to be treated separately and the constant correction term to be adapted.

## 3.4. Discussion

The described modulo $(2^n - 1)$ multiplier is almost as efficient as an ordinary integer multiplier with respect to circuit size and delay, but has an even more regular structure. Booth recoding and Wallace-tree addition can both be applied for speed-up. The $n$-bit modulo final adder is as fast but smaller than the $2n$-bit final adder used for $n$-bit integer multiplication.

The same holds true for the modulo $(2^n + 1)$ multiplier which is slightly less efficient due to the additional correction term and the $2^n$-correction. It is suited for normal and diminished-one number representation. The correction term is constant as opposed to [13], where the precomputation of the data-dependent correction term $Z$ adds a delay of some full-adders (i.e., an $(n - 1)$-bit counter). Compared to [9], two carry-save adder stages for modulo reduction after the carry-save adder array are eliminated.

## 4. Modulo $(2^n + 1)$ multiplication-addition

In the IDEA cipher algorithm, two of the four modulo $(2^n + 1)$ multiplications required for one encryption round are followed immediately by a modulo $2^n$ addition [8]:

$$\begin{aligned} P &= X \cdot Y \bmod (2^n + 1) \\ S &= (P + A) \bmod 2^n \end{aligned} \tag{37}$$

This multiply-add structure is on the critical path of the IDEA data path and should therefore be made as fast as possible. A common speed-up technique is to include the output addition as carry-save addition before the final adder of the multiplier, thus reducing the number of carry-propagation

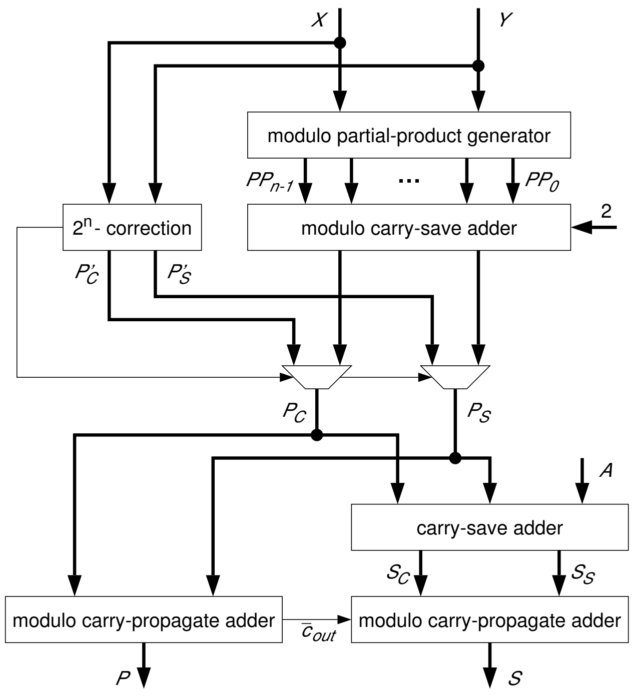

**Figure 9. Modulo $(2^n + 1)$ multiplier-adder architecture.**

steps in series from two to one. Because the product $P$ and the sum $S$ both are used as outputs, two parallel final adders are required. And because the final adder of a modulo multiplier also performs a final modulo correction step (i.e., by adding $\overline{c}_{out}$ according to (20)), the same correction has to be done in both final adders:

$$\begin{aligned} P &= (P_C + P_S + 1) \bmod (2^n + 1) \\ &= (P_C + P_S + \overline{c}_{out}) \bmod 2^n \quad (38) \\ S &= \big((P_C + P_S + 1) \bmod (2^n + 1) + A\big) \bmod 2^n \\ &= (P_C + P_S + \overline{c}_{out} + A) \bmod 2^n \\ &= (S_C + S_S + \overline{c}_{out}) \bmod 2^n \quad (39) \end{aligned}$$

where $S_C + S_S = P_C + P_S + A$ is the carry-save addition of the redundant product $(P_C, P_S)$ and the addend $A$ yielding the redundant sum $(S_C, S_S)$, and $\overline{c}_{out}$ is the inverted carry-out of the final adder for product $P$ (38). Fig. 9 depicts the architecture of the modulo $(2^n + 1)$ multiplier-adder (again, all buses are $n$ bits wide). In this solution, the number of carry propagations has been reduced from four when using standard components (i.e., multiplier final adder, modulo adder-incrementer, and output adder) down to one.

## 5. Implementations and results

A modulo $(2^n - 1)$ adder, a modulo $(2^n + 1)$ adder, and an integer adder have been implemented based on the parallel-prefix adder architecture described in Section 2. A modulo $(2^n - 1)$ multiplier, a modulo $(2^n + 1)$ multiplier and multiplier-adder (denoted as "mod $(2^n + 1)$ +"), and an integer multiplier have been implemented using Wallace trees for carry-save addition and the modulo parallel-prefix adders for final addition, but with no Booth recoding. All units have been described as parameterized circuit generators in synthesizable VHDL code [17].

**Table 3. Unit-gate adder results.**

| adder | area | delay |
|---|---|---|
| integer | $\frac{3}{2}n \log n + 4n$ | $2 \log n + 3$ |
| mod $(2^n - 1)$ | $\frac{3}{2}n \log n + 7n$ | $2 \log n + 5$ |
| mod $(2^n + 1)$ | $\frac{3}{2}n \log n + 7n$ | $2 \log n + 5$ |

**Table 4. Unit-gate multiplier results.**

| multiplier | area | delay |
|---|---|---|
| integer | $8n^2 + 3n \log n - 3n$ | $4d(n) + 2 \log n + 6$ |
| mod $(2^n - 1)$ | $8n^2 + \frac{3}{2}n \log n - 7n$ | $4d(n) + 2 \log n + 6$ |
| mod $(2^n + 1)$ | $9n^2 + \frac{3}{2}n \log n + 11n$ | $4d(n+1) + 2 \log n + 9$ |
| mod $(2^n + 1)+$ | $9n^2 + 2n \log n + 25n$ | $4d(n+1) + 2 \log n + 13$ |

### 5.1. Unit-gate model

Circuit size and delay estimates can be given on a unit-gate basis. Thereby, each two-input monotonic gate (e.g., AND, NAND) counts as one gate (area and delay), an XOR as two gates (area and delay), and a full-adder has an area of seven gates and a delay of four gates. Tables 3 and 4 give the gate-count and gate-delay estimates as a function of the word length $n$ for the adders and multipliers, respectively. Thereby, $d(n)$ denotes the depth of the Wallace tree in multipliers [13]:

$$\begin{array}{rcccccccccc} n &=& 3 & 4 & 5\text{–}6 & 7\text{–}9 & 10\text{–}13 & 14\text{–}19 & 20\text{–}28 & 29\text{–}42 & 43\text{–}63 & \cdots \\ d(n) &=& 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \cdots \end{array}$$

### 5.2. Standard-cell implementation

Circuits have been synthesized from their VHDL descriptions and optimized for highest speed with the synthesis tools by Synopsys, Inc. A 0.25 $\mu$m standard-cell library under typical PTV conditions (i.e., typical process, 25° C, 2.5 V) has been used. For comparison, integer and modulo $(2^n - 1)$ adders and multipliers have been synthesized using standard components from the Synopsys DesignWare Foundation Library (denoted as "DW") with the

**Table 5. Standard-cell adder results.**

| adder | 8 bit | | 16 bit | | 32 bit | | 64 bit | |
|---|---|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay | area | delay |
| integer | 4239 | 0.52 | 7137 | 0.71 | 15336 | 0.93 | 34065 | 1.14 |
| mod $(2^n - 1)$ | 4365 | 0.78 | 10611 | 0.93 | 19269 | 1.17 | 43452 | 1.43 |
| mod $(2^n + 1)$ | 4806 | 0.77 | 8181 | 1.06 | 23706 | 1.16 | 45000 | 1.44 |
| DW integer | 4923 | 0.55 | 15021 | 0.75 | 22608 | 0.89 | 50130 | 1.09 |
| DW $(2^n - 1)$ | 6975 | 0.92 | 14400 | 1.30 | 30771 | 1.58 | 70443 | 1.95 |

**Table 6. Standard-cell multiplier results.**

| multiplier | 8 bit | | 16 bit | | 32 bit | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| integer | 16668 | 2.32 | 61542 | 3.33 | 237564 | 4.51 |
| mod $(2^n - 1)$ | 16740 | 2.54 | 60894 | 3.51 | 233127 | 4.83 |
| mod $(2^n + 1)$ | 20232 | 2.47 | 66213 | 3.60 | 236574 | 4.76 |
| mod $(2^n + 1) +$ | 23256 | 2.91 | 74835 | 3.95 | 258858 | 5.02 |
| DW integer | 18306 | 2.55 | 57690 | 3.46 | 202131 | 4.26 |
| DW mod $(2^n - 1)$ | 22194 | 3.70 | 70902 | 4.98 | 228573 | 6.16 |

fastest circuit architectures (i.e., fast carry-lookahead adder "clf", Booth-Wallace multiplier "wall"). Thereto, modulo addition requires an integer adder and an incrementer (16), while modulo multiplication requires an integer multiplier, an adder, and an incrementer (24). The results are given in Tables 5 and 6. The differences between the custom and the DesignWare integer adders and multipliers are mainly due to the different carry-lookahead adder structures and to Booth recoding (i.e., not used in the custom multipliers). All custom modulo arithmetic units show considerable speed and, in most cases, also area advantages compared to the solutions based on standard components. The proposed modulo $(2^n + 1)$ multiplier-adder allows the implementation of a high-performance IDEA cipher engine delivering up to 720 Mbit/s data rate at 100 MHz clock frequency.

## 6. Conclusions

Parallel-prefix adders with an additional prefix level have been used to implement novel fast and simple end-around-carry adders for modulo $(2^n \pm 1)$ addition. Modulo $(2^n \pm 1)$ multiplication has been realized using modulo-reduced partial products, modulo carry-save adders, and a modulo final adder, resulting in the fastest modulo multiplier circuits reported in the literature. Their architecture allows the use of Wallace-tree addition and Booth recoding of partial products for speed-up. An optimized modulo multiplier-adder has been presented for the efficient circuit implementation of the IDEA block cipher. The performance of all proposed modulo arithmetic units is only slightly inferior to units for ordinary integer addition and multiplication. The highly regular structure of the units allows their description by circuit generators purely in parameterized synthesizable VHDL code, which makes them suitable for efficient implementation of high-performance modulo-arithmetic units in modern cell-based VLSI technologies.

## References

[1] D. P. Agrawal and T. R. N. Rao. Modulo $(2^n + 1)$ arithmetic logic. *IEEE J. on Electronic Circuits and Syst.*, 2:186–188, Nov. 1978.

[2] M. A. Bayoumi, G. A. Jullien, and W. C. Miller. A VLSI implementation of residue adders. *IEEE Trans. Circuits and Syst.*, CAS-34(3):284–288, Mar. 1987.

[3] A. V. Curiger, H. Bonnenberg, and H. Kaeslin. Regular VLSI architectures for multiplication modulo $(2^n + 1)$. *IEEE J. Solid-State Circuits*, 26(7):990–994, July 1991.

[4] C. Efstathiou, D. Nikolos, and J. Kalamatianos. Area-time efficient modulo $2^n - 1$ adder design. *IEEE Trans. Circuits and Syst.*, 41(7):463–467, July 1994.

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

[6] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[7] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, Oct. 1980.

[8] X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology – EUROCRYPT'90*, pages 389–404, Berlin, Germany: Springer-Verlag, 1990.

[9] Y. Ma. A simplified architecture for modulo $(2^n + 1)$ multiplication. *IEEE Trans. Comput.*, 47(3):333–337, Mar. 1998.

[10] A. Skavantzos and P. B. Rao. New multipliers modulo $2^n - 1$. *IEEE Trans. Comput.*, 41(8):957–961, Aug. 1992.

[11] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*. IEEE Press, New York, 1986.

[12] A. Tyagi. A reduced-area scheme for carry-select adders. *IEEE Trans. Comput.*, 42(10):1162–1170, Oct. 1993.

[13] Z. Wang, G. A. Jullien, and W. C. Miller. An efficient tree architecture for modulo $2^n + 1$ multiplication. *J. VLSI Signal Processing Systems*, 14(3):241–248, Dec. 1996.

[14] S. Wei and K. Shimizu. Modulo $2^p - 1$ arithmetic hardware algorithm using signed-digit number representation. *IEICE Trans. Inform. & Systems*, E79-D(3):242–246, Mar. 1996.

[15] R. Zimmermann. Non-heuristic optimization and synthesis of parallel-prefix adders. In *Proc. Int. Workshop on Logic and Architecture Synthesis*, pages 123–132, Grenoble, France, Dec. 1996.

[16] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998.

[17] R. Zimmermann. VHDL library of arithmetic units. In *Proc. 1st Int. Forum on Design Languages (FDL'98)*, Lausanne, Switzerland, Sept. 1998.