Modular Multiplication for FPGA Implementation of the IDEA Block Cipher

Jean-Luc Beuchat

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modular Multiplication for FPGA Implementation of the IDEA Block Cipher

Jean-Luc Beuchat

No 4558

September 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4558--FR+ENG

Modular Multiplication for FPGA Implementation of the IDEA Block Cipher

Jean-Luc Beuchat

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arénaire

Rapport de recherche n ° 4558 — September 2002 — 17 pages

Abstract: The IDEA block cipher is a symmetric-key algorithm which encrypts 64-bit plaintext blocks to 64-bit ciphertext blocks, using a 128-bit secret key. The security of IDEA relies on combining operations from three algebraic groups: integer addition modulo 2^n, bitwise exclusive or of two n-bit words, and integer multiplication modulo $(2^n + 1)$ which is the critical arithmetic operation of the block cipher. In this paper, we investigate three algorithms based on a small multiplication with a subsequent modulo correction. They are particularly well suited for the latest FPGA devices embedding small multiplier blocks, like the Virtex-II family. We also consider a multiplier based on modulo $(2^n + 1)$ adders. Several architectures of the IDEA block cipher are then described and compared from different point of view: throughput to area ratio or adequation with feedback and non-feedback chaining modes. Our fastest circuit achieves a throughput of 8.5 Gb/s, which is, to our knowledge, the best rate reported in the literature.

Key-words: Computer arithmetic, modulo $(2^n + 1)$ multiplication, IDEA block cipher, cryptography, FPGA

(Résumé : tsvp)

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International : +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International : +33 4 76 61 52 52

Multiplication modulaire pour l'implantation sur FPGA de l'algorithme de cryptage par bloc IDEA

Résumé : IDEA est un algorithme de cryptage à clef privée transformant un bloc de 64 bits de texte en clair en un bloc de 64 bits de texte chiffré. La sécurité de IDEA repose sur la combinaison d'opérations dans trois groupes : l'addition modulo 2^n, le ou exclusif bit à bit de mots de n bits et la multiplication modulo $(2^n + 1)$. Cette dernière est l'opération arithmétique critique d'IDEA. Dans cet article, nous étudions trois algorithmes exploitant de petites multiplications suivies d'une correction. Ils sont particulièrement bien adaptés aux circuits FPGA récents contenant de petits multiplieurs, comme la famille Virtex-II. Nous considérons également un opérateur basé sur des additions modulo $(2^n + 1)$. Plusieurs architectures d'un processeur IDEA sont ensuite décrites et évaluées de différents points de vue : relation entre la surface et le débit ou adéquation avec les divers modes de chaînage. Notre circuit le plus rapide atteint un débit de 8.5 Gb/s, ce qui est à notre connaissance le meilleur résultat publié à ce jour.

Mots-clé : Arithmétique des ordinateurs, multiplication modulo $(2^n + 1)$, IDEA, cryptographie, FPGA

1 Introduction

The IDEA (International Data Encryption Algorithm) block cipher [8] is a symmetric-key algorithm which encrypts 64-bit plaintext blocks to 64-bit ciphertext blocks, using a 128-bit key K. IDEA consists in 8 computationally identical rounds followed by an output transformation (Figure 1). Round r ($1 \leq r \leq 8$) transforms a 64-bit input into four 16-bit output blocks forming the input of the next round. The computation involves six 16-bit subkeys $K_i^{(r)}$ ($1 \leq i \leq 6$) derived from K. The output transformation, employing four additional subkeys $K_i^{(9)}$ ($1 \leq i \leq 4$), produces the ciphertext. The security of IDEA relies on combining operations from three algebraic groups. The three group operations on n-bit words x and y ($n = 16$) are:

- Integer addition modulo 2^n (denoted by $x \boxplus y$).
- Bitwise exclusive or (denoted by $x \oplus y$).
- Modified integer multiplication modulo $(2^n + 1)$, with $0 \in \mathbb{Z}_{2^n}$ associated with $2^n \in \mathbb{Z}^*_{2^n+1}$ (denoted by $x \odot y$). If $x = 0$ or $y = 0$, we replace it by 2^{16}. If the result of the multiplication is 2^{16}, it is replaced by 0.

Decryption is achieved using the same algorithm with the ciphertext provided as input. The only difference lies in the key schedule generating the $K_i^{(r)}$ coefficients.

IDEA encrypts plaintext in fixed-size 64-bit blocks. However, messages will often exceed 64 bits and a simple solution, known as Electronic Codebook (ECB) mode, consists in partitioning the plaintext into 64-bit blocks and encrypting each independently. This ECB mode has however a drawback in the sense that identical ciphertext blocks imply identical plaintext blocks and is therefore discommended if the secret key is reused for more than one message. More sophisticated chaining modes bring a solution to this problem. For instance, in the Cipher Block Chaining (CBC) mode, a feedback mechanism causes the jth ciphertext block to depend on the first j plaintext blocks and an n-bit initialization vector. Since the entire dependency on preceding blocks is contained in the previous ciphertext block [12], all blocks must be processed sequentially[1]. This property forbids to pipeline the computation path and implies a slightly different hardware architecture of the block cipher with a lower throughput. The counter (CTR) mode, a non-feedback mode described for example in [5], could remedy the situation if it becomes a standard as recommended in [10]. It is also possible to pipeline the processor in feedback modes if we accept the decomposition of the data stream into d separately encrypted messages, where d is the pipeline depth [4].

Multiplication modulo $(2^n + 1)$ is the critical arithmetic operation of this block cipher: both area and speed of an IDEA processor are strongly related to the hardware operator carrying out $x \odot y$. The main purpose of this paper is to design efficient arithmetic operators for FPGA based implementations of IDEA. After a brief overview of Virtex-E and Virtex-II FPGA families (Section 2), we investigate four algorithms dedicated to the $\odot$ operator (Section 3). In order to easily compare several architectures, we have developed a tool which generates the synthesizable VHDL code of an IDEA processor. Several parameters allow us to choose one of the multipliers described in Section 3 and the latency of the three operators $\odot$, $\boxplus$, and $\oplus$ (respectively denoted by α, β, and γ on Figure 1). We describe this tool in Section 4. Finally, Section 5 digests our main results and compares them with recent works on IDEA.

2 Some Features of the Virtex-E and Virtex-II Families

This section gives a brief overview of useful features of Virtex-E and Virtex-II devices for this work. Virtex-E and Virtex-II configurable logic blocks (CLBs) provide functional elements for synchronous and combinatorial logic. Each CLB includes respectively two (Virtex-E) or four (Virtex-II) slices containing basically two 4-input look-up tables (LUT), two storage elements, and fast

[1] CBC decryption can however be performed in parallel.

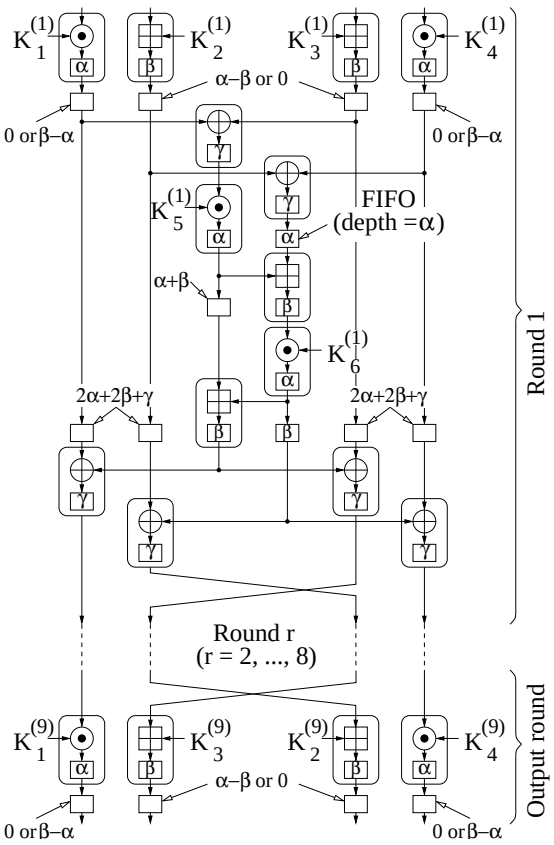

Figure 1: IDEA computation path and its optional pipeline stages.

carry logic dedicated to addition and subtraction (Figure 2a).

A Virtex-II device also embeds many 18×18 two's complement multipliers (the MULT18x18 blocks), each of them supporting two input ports 18-bit signed or 17-bit unsigned wide. This width is ideal for the implementation of the modulo $(2^{16}+1)$ multiplication. Furthermore, each multiplier has an optional internal pipeline stage. Surprisingly, this feature is poorly documented in the Virtex-II data sheet and synthesis tools seem unable to automatically deal with it. The MULT18x18S component, available in the library of Synplify Pro, allows us to write multipliers that take advantage of this characteristic (Figure 2c).

Other FPGA families (ORCA series 4 from Lattice or Stratix from Altera) also embed small multipliers. Their width is smaller than 16 and it is therefore necessary to effectively build the required multiplier from these blocks (see for instance [3]).

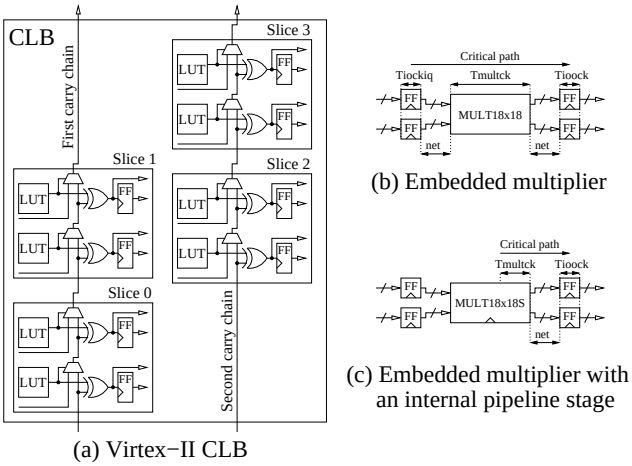

Figure 2: Virtex-II arithmetic features overview.

3 Modular Multiplication

Designing an area-time efficient modulo (2^n+1) multiplier is the key point of the hardware implementation and depends on the target FPGA resources. The first three algorithms involve small multipliers and are thus particularly dedicated to Virtex-II devices (paragraphs 3.1 to 3.3). We also consider a multiplier based on modulo (2^n+1) adders (paragraph 3.4) proposed by R. Zimmermann for VLSI implementations [14]. Such an operator requires only LUTs and could be an interesting alternative for the Virtex-E family.

3.1 Low-High Algorithm

The Low-High algorithm was originally described by the designer of the IDEA block cipher [8]. This algorithm, defined by Equation (1), provides the programmer with a tool to perform modulo (2^n+1) multiplication when $x, y \in \mathbb{Z}^*_{2^n+1}$.

$$x \odot y = \begin{cases} xy \bmod 2^n - xy \operatorname{div} 2^n + 2^n + 1 \\ \quad \text{if } xy \operatorname{div} 2^n > xy \bmod 2^n, \\ xy \bmod 2^n - xy \operatorname{div} 2^n \\ \quad \text{if } xy \operatorname{div} 2^n \leq xy \bmod 2^n. \end{cases} \quad (1)$$

However, the cases where $x = 0$ or $y = 0$ must be handled separately. Note that

$$\begin{aligned} (2^n \cdot j) \bmod (2^n+1) &= (-j) \bmod (2^n+1) \\ &= 2^n + 1 - j, \end{aligned} \quad (2)$$

where $1 \leq j \leq 2^n$. Assume now that $y = 0$: due to this special encoding of 2^n, Equation (2) becomes

$$x \odot y = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{if } x = 1, \\ 2^n + 1 - x & \text{otherwise.} \end{cases}$$

Consequently,

$$x \odot y = \begin{cases} (2^n + 1 - x) \bmod 2^n & \text{if } y = 0, \\ (2^n + 1 - y) \bmod 2^n & \text{if } x = 0. \end{cases}$$

Figure 3 depicts the hardware architecture of a modulo (2^n+1) multiplication operator based on the Low-High algorithm. It primarily consists in an $n \times n$ unsigned multiplier and a modulo (2^n+1) subtracter whose inputs c_L and c_H are selected by a multiplexer depending on x and y:

$$c_L = \begin{cases} (2^n + 1 - x) \bmod 2^n & \text{if } y = 0, \\ (2^n + 1 - y) \bmod 2^n & \text{if } x = 0, \\ xy \bmod 2^n & \text{if } x \neq 0 \text{ and } y \neq 0, \end{cases} \quad (3)$$

and

$$c_H = \begin{cases} 0 & \text{if } y = 0, \\ 0 & \text{if } x = 0, \\ xy \operatorname{div} 2^n & \text{if } x \neq 0 \text{ and } y \neq 0. \end{cases} \quad (4)$$

Replacing (3) and (4) in (1) yields

$$x \odot y = \begin{cases} (c_L - c_H + 1) \bmod 2^n & \text{if } c_H > c_L, \\ (c_L - c_H) \bmod 2^n & \text{if } c_H \leq c_L. \end{cases} \quad (5)$$

Our VHDL generator allows the user to shorten the critical path by inserting pipeline stages in the multiplier (integer parameter m_1), after the subtracter (boolean parameter m_2), and after the final adder (boolean parameter m_3). The parameter $m_1 \in \{0, 1, 2, 3\}$ requires further explanations. Since MULT18x18 blocks are not available in Virtex-E devices, the 16×16 multiplier is implemented in logic (CLBs and fast carry logic). Synthesis tools are generally able to pipeline automatically such a circuit by moving m_1 register stages into the multiplier. Therefore, up to three pipeline stages are available in our implementation. Since the small embedded multipliers of Virtex-II devices have an optional internal pipeline stage, m_1 is equal to 0 or 1 for this family.

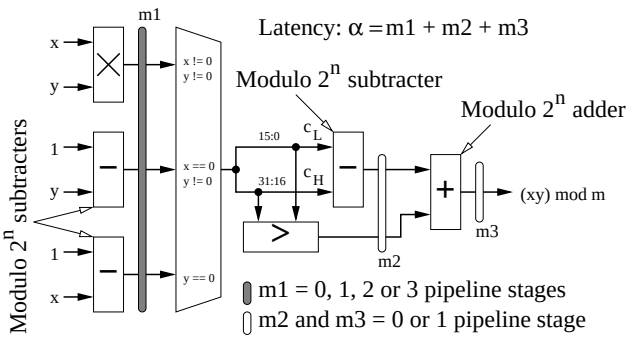

Figure 3: Modulo (2^n+1) multiplier based on the Low-High algorithm.

Example 3.1
Let us illustrate the behavior of this circuit for some examples:

- *For $x = y = 0$, we have $c_L = 1$ and $c_H = 0$. As $c_H \leq c_L$, $x \odot y = c_L - c_H = 1$.*

- *For $x = 0$ and $y = 1$, we have $c_L = c_H = 0$ and $x \odot y = 0$. Since the value 2^n is replaced by 0, this result is correct ($2^n \ mod \ (2^n + 1) = 2^n$).*

- *For $x = 10$ and $y = 9$, we obtain $c_L = 90$ and $c_H = 0$. As $c_H \leq c_L$, $x \odot y = c_L - c_H = 90$.*

- *For $x = 16384$ and $y = 8$, we have $c_L = 0$ and $c_H = 2$. Consequently, $x \odot y = (c_L - c_H + 1) \ mod \ 2^n = 65535$.*

3.2 Low-High Algorithm Revisited

As described above, the Low-High algorithm involves inter alia the comparison of two 16-bit integers and two subtractions. A clever rewriting allows us to get rid of the subtractions at the price of a larger multiplexer and to remove the comparator. Assume that x and $y \in \{1, \ldots, 2^n - 1\}$. We have

$$\begin{aligned} & xy \bmod 2^n \geq xy \operatorname{div} 2^n \\ & \Leftrightarrow xy \bmod 2^n - xy \operatorname{div} 2^n \geq 0 \\ & \Leftrightarrow xy \bmod 2^n + \underbrace{2^n - xy \operatorname{div} 2^n}_{\text{Two's complement}} \geq 2^n \\ & \Leftrightarrow xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 1 \geq 2^n, \end{aligned} \tag{6}$$

and

$$\begin{aligned} & (xy \bmod 2^n - xy \operatorname{div} 2^n) \bmod 2^n \\ & \quad = (xy \bmod 2^n + 2^n - xy \operatorname{div} 2^n) \bmod 2^n \\ & \quad = (xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 1) \bmod 2^n. \end{aligned} \tag{7}$$

Replacing (6) and (7) in (5) yields

$$x \odot y = \begin{cases} (xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 1) \bmod 2^n \\ \quad \text{if } xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 1 \geq 2^n, \\ (xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 2) \bmod 2^n \\ \quad \text{if } xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 1 < 2^n, \end{cases}$$

and the comparator is no longer necessary. The removal of the two subtracters is straightforward: we rewrite Equations (3) and (4) as

$$c_L = \begin{cases} 0 & \text{if } x \neq 0 \text{ and } y = 0, \\ 0 & \text{if } x = 0 \text{ and } y \neq 0, \\ 1 & \text{if } x = 0 \text{ and } y = 0, \\ xy \bmod 2^n & \text{otherwise,} \end{cases}$$

and

$$c_H = \begin{cases} x & \text{if } x \neq 0 \text{ and } y = 0, \\ y & \text{if } x = 0 \text{ and } y \neq 0, \\ 0 & \text{if } x = 0 \text{ and } y = 0, \\ xy \operatorname{div} 2^n & \text{otherwise.} \end{cases}$$

Figure 4 illustrates the corresponding hardware operator (the parameters m_1, m_2 and m_3 are identical to those described in Section 3.1).

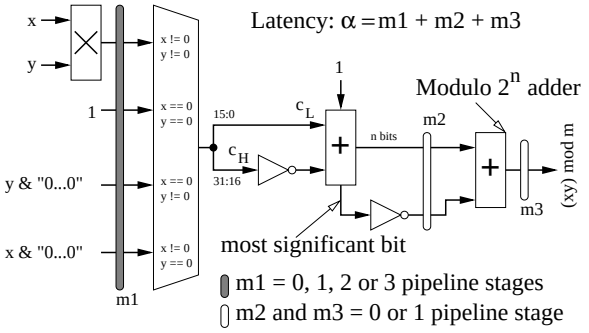

Figure 4: Modulo (2^n+1) multiplier based on the modified Low-High algorithm.

Example 3.2
Let us illustrate the behavior of our second circuit for the examples already studied above:

- *For $x = y = 0$, we have $c_L = 1$, $c_H = 0$, and $\overline{c_H} = 2^n - 1$. As $c_L + \overline{c_H} + 1 = 2^n + 1$, $x \odot y = (2^n + 1) \ mod \ 2^n = 1$.*

- *For $x = 0$ and $y = 1$, we have $c_L = 0$, $c_H = y = 1$, and $\overline{c_H} = 2^n - 2$. Consequently, $c_L + \overline{c_H} + 1 = 2^n - 1$ and $x \odot y = (c_L + \overline{c_H} + 2) \ mod \ 2^n = 0$. Since the value 2^n is replaced by 0, this result is correct.*

- *For* $x = 10$ *and* $y = 9$, *we obtain* $c_L = 90$, $c_H = 0$, *and* $\overline{c_H} = 2^n - 1$. *As* $c_L + \overline{c_H} + 1 = 90 + 2^n$, $x \odot y = (c_L + \overline{c_H} + 1)\ mod\, 2^n = 90$.
- *For* $x = 16384$ *and* $y = 8$, *we have* $c_L = 0$, $c_H = 2$, *and* $\overline{c_H} = 2^n - 3$. *Consequently,* $c_L + \overline{c_H} + 1$ *is equal to* $2^n - 2$ *which is strictly smaller than* 2^n *and* $x \odot y = (c_L + \overline{c_H} + 2)\ mod\, 2^n = 2^n - 1 = 65535$.

3.3 (n+1)×(n+1) Multiplier Based Operator

The operator depicted on Figure 4 still comprises a multiplexer to handle the special cases where $x = 0$ or $y = 0$. We describe here how to treat correctly these cases with an $(n+1) \times (n+1)$ multiplier. Let us define two $(n+1)$-bit integers $\tilde{x}$ and $\tilde{y}$ such as

$$\tilde{x} = \begin{cases} x & \text{if } 1 \le x \le 2^n - 1 \\ 2^n & \text{if } x = 0 \end{cases}$$

and

$$\tilde{y} = \begin{cases} y & \text{if } 1 \le y \le 2^n - 1 \\ 2^n & \text{if } y = 0 \end{cases}$$

Modulo $(2^n + 1)$ multiplication is then expressed as

$$\begin{aligned} &\tilde{x}\tilde{y} \bmod (2^n + 1) \\ &\quad = (M + 2^n D) \bmod (2^n + 1) \\ &\quad = \left(M + d_n 2^{2n} + 2^n \cdot \sum_{i=0}^{n-1} d_i 2^i \right) \bmod (2^n + 1), \end{aligned} \quad (8)$$

where

$$M = \tilde{x}\tilde{y} \bmod 2^n \quad \text{and} \quad D = \tilde{x}\tilde{y} \operatorname{div} 2^n = \sum_{i=0}^{n} d_i 2^i.$$

Substituting

$$2^n \bmod (2^n + 1) = (-1) \bmod (2^n + 1)$$

and

$$\begin{aligned} &2^{2n} \bmod (2^n + 1) \\ &\quad = ((2^n \bmod (2^n + 1)) \cdot (2^n \bmod (2^n + 1))) \bmod (2^n + 1) \\ &\quad = ((-1) \cdot (-1)) \bmod (2^n + 1) = 1, \end{aligned}$$

into (8) yields

$$\begin{aligned} &\tilde{x}\tilde{y} \bmod (2^n + 1) \\ &\quad = \left(M + d_n - \sum_{i=0}^{n-1} d_i 2^i \right) \bmod (2^n + 1) \\ &\quad = \left(M + d_n + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \bmod (2^n + 1). \end{aligned} \quad (9)$$

Remember now that a modulo m reduction is defined by

$$(km \le x < (k+1)m) \Leftrightarrow (x \bmod m = x - km). \quad (10)$$

We apply (10) to (9) and obtain a modulo $(2^n + 1)$ multiplication algorithm for $(n+1)$-bit integers belonging to $\mathbb{Z}^*_{2^n+1}$:

$$\begin{aligned} &\tilde{x}\tilde{y} \bmod (2^n + 1) \\ &\quad = \begin{cases} M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 - 2^n - 1 \\ \qquad \text{if } M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \ge 2^n + 1 \\ M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \\ \qquad \text{if } M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 < 2^n + 1 \end{cases} \\ &\quad = \begin{cases} \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 \right) \bmod 2^n \\ \qquad \text{if } M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 \ge 2^n \\ M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \\ \qquad \text{if } M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 < 2^n \end{cases} \end{aligned} \quad (11)$$

A small modification of Equation (11) is still required to perform the modified modulo $(2^n + 1)$ multiplication of IDEA. Remember that $x \odot y = 0$ for $x =$

$0, y = 1$ and $x = 1, y = 0$ ($(2^n \cdot 1) \bmod (2^n + 1) = 2^n$). We have $D = 0$, $M = 1$, and $M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 = 2^n - 1$. Consequently, when $M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 < 2^n$, we obtain

$$x \odot y = \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \bmod 2^n$$
$$= \begin{cases} 0 & \text{if } (x = 0, y = 1), \\ 0 & \text{if } (x = 1, y = 0), \\ M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 & \text{otherwise.} \end{cases}$$

Finally, for $x = y = 0$, we have $d_n = 1$, $M = 0$, and $d_i = 0 \ \forall i \neq n$. Therefore

$$\left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \bmod 2^n = \left(\sum_{i=0}^{n-1} 2^i + 2 \right) \bmod 2^n$$
$$= (2^n + 1) \bmod 2^n = 1$$
$$= x \odot y.$$

We obtain the modulo $(2^n + 1)$ multiplication operator described by Algorithm 3.1 and illustrated on Figure 5.

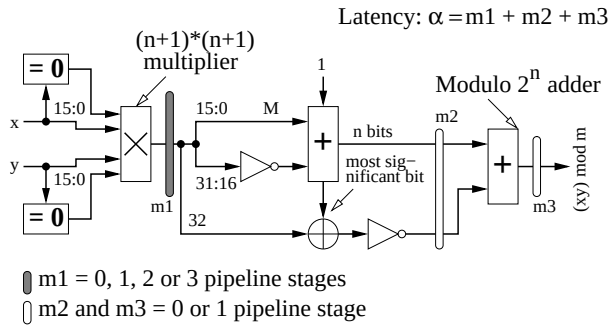

Figure 5: Modulo $(2^n + 1)$ multiplier based on an $(n + 1) \times (n + 1)$ multiplier.

Example 3.3
Let us illustrate the behavior of this third circuit for our four examples:

Algorithm 3.1 Modulo $(2^n + 1)$ multiplication.

1: **if** $x = 0$ **then**
2: $\quad x \leftarrow 2^n$
3: **end if**
4: **if** $y = 0$ **then**
5: $\quad y \leftarrow 2^n$
6: **end if**
7: $M \leftarrow xy \bmod 2^n$ (n-bit integer)
8: $D \leftarrow xy \operatorname{div} 2^n$ ($(n+1)$-bit integer)
9: **if** $\left(d_n = 0 \text{ and } M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 < 2^n \right)$ or $(d_n = 1)$ **then**
10: $\quad x \odot y \leftarrow \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \bmod 2^n$
11: **else**
12: $\quad x \odot y \leftarrow \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 \right) \bmod 2^n$
13: **end if**

- *For* $x = y = 0$, *we have* $M = 0$, $d_n = 1$, *and* $d_i = 0 \ \forall i \neq n$. *We obtain* $x \odot y = \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \ mod \, 2^n = (2^n + 1) \ mod \, 2^n = 1$.
- *For* $x = 0$ *and* $y = 1$, *we have* $M = 0$, $d_0 = 1$, *and* $d_i = 0 \ \forall i \neq 0$. *As* $M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 = 2^n - 1$, *the product is defined by* $x \odot y = \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \ mod \, 2^n = 2^n \ mod \, 2^n = 0$.
- *For* $x = 10$ *and* $y = 9$, *we obtain* $M = 90$, $D = 0$, *and* $M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 = 2^n + 90$. *Consequently,* $x \odot y = \left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 \right) \ mod \, 2^n = 90$.
- *For* $x = 16384$ *and* $y = 8$, *we have* $M = 0$, $D = 2$, *and* $M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 1 = 2^n - 2$. $x \odot y$ *is then equal to* $\left(M + \sum_{i=0}^{n-1} \bar{d}_i 2^i + 2 \right) \ mod \, 2^n = 2^n - 1 = 65535$.

A very close architecture has been published in 1991 by Curiger *et al.* [4]. Instead of the *not xor* gate, they

use an *or* gate to compute the input carry of the second adder. This operator seems however mistaken. For instance, if $x = y = 1$ we have $xy \bmod 2^n = 1$, $xy \operatorname{div} 2^n = 0$, and

$$xy \bmod 2^n + \overline{xy \operatorname{div} 2^n} + 1 = 1\underbrace{00\ldots00}_{(n-1)\times}1.$$

The output carry of the first modulo 2^n adder is therefore equal to one and we obtain

$$xy \bmod (2^n + 1) = (1 + 1) \bmod 2^n = 2$$

which is clearly wrong. This error occurs for several other input values.

3.4 Carry-save Adder Based Architecture

R. Zimmermann has proposed modulo $(2^n \pm 1)$ multiplication operators using modulo-reduced partial products, modulo carry-save adders, and a modulo final adder [14]. Modulo $(2^n + 1)$ multiplication is defined by:

$$xy \bmod (2^n + 1) = \left(n + 2 + \sum_{i=0}^{n-1} \mathrm{PP}_i\right) \bmod (2^n + 1), \qquad (12)$$

where

$$\mathrm{PP}_i = x_i \cdot y_{n-i-1} \cdots y_0 \bar{y}_{n-1} \cdots \bar{y}_{n-i} + \bar{x}_i \cdot 0 \cdots 01 \cdots 1. \qquad (13)$$

Figure 6 depicts the architecture of this modulo $(2^n + 1)$ multiplier and its four optional pipeline stages. A first stage implements Equation (13) to compute the 16 modulo-reduced partial products. Modulo (2^n+1) multiplication is then carried out by summing these terms and the constant 2 with n modulo $(2^n + 1)$ adders. Remember that this operation can be realized by an end-around-carry adder with $c_{\mathrm{in}} = \bar{c}_{\mathrm{out}}$:

$$(a + b + 1) \bmod (2^n + 1) = (a + b + \bar{c}_{\mathrm{out}}) \bmod 2^n. \quad (14)$$

Note that this operator computes the sum $a + b$ increased by one. Since the circuit on Figure 6 involves n modulo $(2^n + 1)$ additions, the term n found in Equation (12) is automatically summed.

As the value 2^n is represented by 0, a 2^n correction unit is required to handle these special cases. R. Zimmermann has defined

$$(C^\star, S^\star) = \begin{cases} (\bar{Y}, 1) & \text{if } X = 2^n \text{ and } Y \neq 2^n, \\ (\bar{X}, 1) & \text{if } X \neq 2^n \text{ and } Y = 2^n, \\ (0, 0) & \text{if } X = Y = 2^n. \end{cases}$$

A multiplexer selects the input of the final modulo $(2^n + 1)$ adder according to x and y. Note that R. Zimmermann has designed an end-around-carry parallel-prefix adder structure to perform this last addition. Prefix adders are however rather inefficient for current FPGA devices and our final modulo $(2^n + 1)$ is based on the following rewriting of Equation (14):

$$(a + b + 1) \bmod (2^n + 1) = \begin{cases} (a + b + 1) \bmod 2^n & \text{if } a + b \geq 2^n, \\ a + b & \text{if } a + b < 2^n. \end{cases}$$

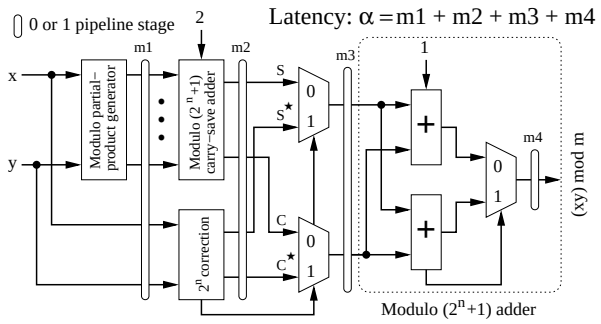

Figure 6: Modulo $(2^n + 1)$ multiplier based on R. Zimmermann's algorithm.

3.5 Comparison of the four algorithms

Table 1 summarizes the main specificities (area and delay) of the four algorithms described above for several sets of parameters. The VHDL code was automatically generated, synthesized using Synplify Pro 7.0.3, and implemented on Virtex-E and Virtex-II devices employing Xilinx Alliance Series 4.1.03i.

For Virtex-II devices, the third modulo (2^n+1) multiplication algorithm allows a significant gain in terms of slices compared to the two Low-High algorithms. This gain is of course less important for Virtex-E devices where the multiplier is implemented on CLBs and requires the main part of the hardware resources. For this family, we also observe a significant increase of the delay between algorithms 2 and 3. This gap is certainly related to the architecture of the $(n+1) \times (n+1)$ multiplier automatically generated by the synthesis tools (one more stage in the tree summing the partial products). Finally, our experiments show that the best set of parameters is $m_1 = 2$, $m_2 = 1$, and $m_3 = 1$ if we want to optimize the throughput of the $\odot$ operator.

The carry-save adder based operator leads to the largest and slowest circuits. Since there is no carry propagation in carry-save adders, synthesis tools don't use dedicated carry logic lines and allocate two LUTs to build a full adder cell. A very low-level VHDL description allows us to take advantage of this carry logic and therefore to reduce the area. However, the routing becomes more complicated and induces an increase in the critical path. These results tend to establish that algorithms intended for ASICs are not always adapted to FPGAs.

4 Architectures of an IDEA processor

Now that we have efficient modulo $(2^n + 1)$ multipliers, the design of a cipher round is straightforward: integer addition modulo 2^n takes advantage of fast-carry logic and LUTs implement bitwise exclusive or. In order to shorten the critical path, each operator has a parametric number of internal pipeline stages (Table 2). Our VHDL generator automatically adds registers in each round in order to correctly synchronize the data according to these parameters (Figure 1). Furthermore, our tool is able to insert registers on the boundaries between two successive rounds.

Table 2: Number of internal pipeline stages of the three group operations.

$\odot$		$\boxplus$	$\oplus$
Algo 1, 2, and 3	Algo 4		
$\alpha \in \{0, \ldots, 3\}$	$\alpha \in \{0, \ldots, 4\}$	$\beta \in \{0, 1\}$	$\gamma \in \{0, 1\}$

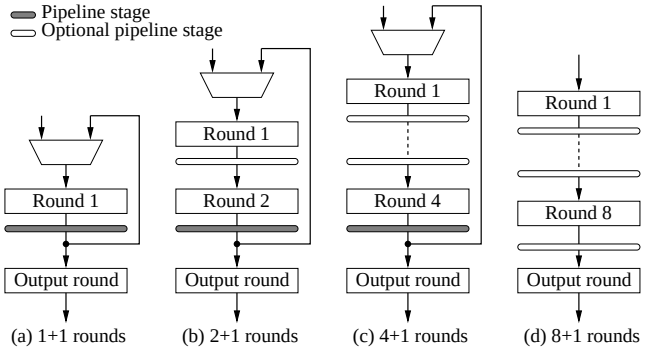

Figure 7: Four architectures suitable for an IDEA processor.

Figure 7a depicts a basic hardware architecture made up of one round, the output transformation, a register, and a multiplexer. A plaintext block is supplied to the circuit through the multiplexer, the first round of IDEA is evaluated, and the result is stored in the register, whose content is then fed back to the hardware round. If the round is implemented as a combinational circuit (i.e. $\alpha = \beta = \gamma = 0$), this basic iterative architecture is tailored to feedback chaining modes like CBC: only one plaintext block is encrypted at a time and we can provide a new input block after nine clock cycles (eight rounds and the output transformation). Let f denote the clock rate of

Table 1: Comparison of the four modulo $(2^n + 1)$ multipliers for Virtex-E and Virtex-II devices.

Device	**Algorithm**	$\mathbf{m_1}$	$\mathbf{m_2}$	$\mathbf{m_3}$	$\mathbf{m_4}$	**Slices**	**Mult. blocks**	**Delay [ns]**
XC2V40-6	Low-High	0	0	0	–	66	1	15.7
XC2V40-6	Improved Low-High	0	0	0	–	48	1	15.9
XC2V40-6	$(n+1) \times (n+1)$ multiplier	0	0	0	–	25	1	15.8
XC2V250-6	Carry-save adder based architecture	0	0	0	0	393	0	15.7
XC2V40-6	Low-High	1	1	1	–	75	1	6.9
XC2V40-6	Improved Low-High	1	1	1	–	58	1	7.1
XC2V40-6	$(n+1) \times (n+1)$ multiplier	1	1	1	–	25	1	6.3
XC2V250-6	Carry-save adder based architecture	1	1	1	1	474	0	6.9
XCV100E-6	Low-High	0	0	0	–	241	–	24.7
XCV100E-6	Improved Low-High	0	0	0	–	204	–	24.3
XCV100E-6	$(n+1) \times (n+1)$ multiplier	0	0	0	–	178	–	32.1
XCV100E-6	Carry-save adder based architecture	0	0	0	–	392	–	31.1
XCV100E-6	Low-High	1	1	1	–	241	–	15.4
XCV100E-6	Improved Low-High	1	1	1	–	226	–	15.2
XCV100E-6	$(n+1) \times (n+1)$ multiplier	1	1	1	–	177	–	22.5
XCV100E-6	Carry-save adder based architecture	1	1	1	1	466	–	11.9
XCV100E-6	Low-High	2	1	1	–	325	–	10.9
XCV100E-6	Improved Low-High	2	1	1	–	318	–	10.8
XCV100E-6	$(n+1) \times (n+1)$ multiplier	2	1	1	–	305	–	15.2
XCV100E-6	Low-High	3	1	1	–	371	–	11.0
XCV100E-6	Improved Low-High	3	1	1	–	365	–	10.8
XCV100E-6	$(n+1) \times (n+1)$ multiplier	3	1	1	–	314	–	14.6

this IDEA processor; its throughput is then defined by

$$\begin{aligned}\text{Throughput}_{(1+1),\text{feedback}} &= \frac{\text{plaintext block size}}{\#\text{rounds}} \cdot f \\ &= \frac{64}{9} f.\end{aligned}$$

Note that for a non-feedback chaining we can provide a new block as soon as the preceding one enters the output transformation (i.e. after eight clock cycles). The throughput is then

$$\text{Throughput}_{(1+1),\text{non-feedback}} = 8f.$$

Figures 7b and 7b illustrates two architectures with partial loop unrolling. For combinational rounds, the critical path increases proportionally to the number of unrolled rounds. This approach is therefore not recommended for feedback chaining modes. After insertion of pipeline stages, it achieves good encryption rates (in non-feedback modes) on small devices.

Finally, Figure 7d shows an architecture with full loop unrolling dedicated to high throughput implementations of the block cipher. The throughput is now

$$\begin{aligned}\text{Throughput}_{(8+1),\text{non-feedback}} &= \text{plaintext block size} \cdot f \\ &= 64f.\end{aligned}$$

In addition to the IDEA computation path, each processor contains a subkey memory implemented on CLBs (roughly 450 slices) and a control unit. The latter simply consists in a token associated with each plaintext block. In addition to indicating the validity of the data, the token selects the correct subkeys in iterative architectures.

5 Results

5.1 Experimental Setup

All experiments described in this paper were performed on a Sun Microsystems Ultra-10 workstation (440 MHz, 1 GB of memory). In addition to the IDEA computation path, each processor contains a subkey memory and a token based control unit. Furthermore, all input and output signals are routed through the D-type flip-flops available in the Input/Output blocks of Virtex-E or Virtex-II devices.

The VHDL code was generated by our tools, synthesized using Synplify Pro 7.0.3 and implemented on Virtex-E and Virtex-II devices employing Xilinx Alliance Series 4.1.03i. The required time ranges from 10 minutes (4+1 rounds, $(n+1)\times(n+1)$ multiplier based operator, XC2V500) to 8 hours (8+1 rounds, carry-save adder based operator, XCV2000E).

5.2 Non-Feedback Chaining Modes

Let us study first some architectures designed for non-feedback chaining modes. Table 3 digests the main characteristics of IDEA processors performing the modified modulo (2^n+1) multiplication with a small multiplier and a subsequent modulo correction (algorithms 1 to 3). Processors working with the third algorithm are clearly the most interesting for Virtex-II devices: as they approximately require 60% of the slices, we can implement a chaining mode or the subkey schedule on the same FPGA. Note that the pipeline contains 107 stages on XC2V1000 and XCV1000E devices (8+1 rounds) and 55 stages on XC2V500 devices (4+1 rounds). Consequently, we should decompose the data stream in 107 or 55 separate messages in order to use these processors in a feedback mode.

The second algorithm offers the best trade-off between area and encryption rate on Virtex-E devices. Although adding a fourth pipeline stage in the multiplier improves frequency, the resulting processor requires a larger and more expensive device (Table 4). This result is surprising and probably comes from the retiming algorithm of the synthesis tools. We have modified our code generator to describe explicitly how the multiplier is to be implemented. We sum up the n partial products $P_i = x_i y$, $i \in \{0,\ldots,n-1\}$ with a tree of carry-propagate adders (Figure 8). The difficulty lies in the determination of the width of the intermediate sums.

Table 3: Characteristics of our architectures for the three multiplier based $\odot$ operators (no register on boundaries between two successive rounds, $m_1 = m_2 = m_3 = 1$, $\beta = 1$, and $\gamma = 1$).

Device	Algorithm	Rounds	Slices	Mult. blocks	Delay [ns]	Through-put [Gb/s]
XC2V1000-6	Low-High	8+1	4845 (94%)	34	8.0	~ 8.0
XC2V1000-6	Improved Low-High	8+1	4199 (82%)	34	8.1	~ 7.9
XC2V1000-6	$(n+1) \times (n+1)$ multiplier	8+1	3077 (60%)	34	7.5	~ 8.5
XC2V500-6	Low-High	4+1	2905 (94%)	18	7.9	~ 4.0
XC2V500-6	Improved Low-High	4+1	2436 (79%)	18	8.0	~ 4.0
XC2V500-6	$(n+1) \times (n+1)$ multiplier	4+1	1983 (64%)	18	7.7	~ 4.1
XCV1000E-6	Low-High	8+1	10024 (81%)	–	14.9	~ 4.3
XCV1000E-6	Improved Low-High	8+1	9586 (78%)	–	14.9	~ 4.3
XCV1000E-6	$(n+1) \times (n+1)$ multiplier	8+1	8745 (71%)	–	22.9	~ 2.8

Note that

$$\begin{aligned} \max(2P_{i+1} + P_i) &= \sum_{j=0}^{n-1} 2^j + 2 \cdot \sum_{j=0}^{n-1} 2^j \\ &= 2^n - 1 + 2 \cdot (2^n - 1) \\ &= 2^{n+1} + 2^n - 3. \end{aligned} \tag{15}$$

Consequently, the sum of two n-bit partial products is an $(n+2)$-bit number. We deduce from (15) that

$$\begin{aligned} \max &= (2^2 \cdot \underbrace{(2P_{i+3} + P_{i+2})}_{n+2 \text{ bits}} + \underbrace{(2P_{i+1} + P_i)}_{n+2 \text{ bits}}) \\ &= 2^2 \cdot (2^{n+1} + 2^n - 3) + 2^{n+1} + 2^n - 3 \\ &= 2^{n+3} + 2^{n+2} + 2^{n+1} + 2^n - 15, \end{aligned}$$

which is an $(n+4)$-bit number. Our VHDL generator applies such rules to build the 16×16 multiplier. The new combinational modulo $(2^n + 1)$ multiplier requires 183 slices compared to 204 for the primary circuit. The measured delay of the circuit is 29 ns. The benefit of this approach is more obvious for sequential operators: 244 slices and 9.2 ns compared to 318 slices and 10.8 ns for the same latency of four clock cycles. Thanks to this operator, the IDEA processor with full loop unrolling fits into one XCV1000E FPGA. This experiment illustrates the inefficiency of the actual synthesis tools for the design of arithmetic operators. The design of new tools should become an important research field within the next years.

Finally, the carry-save adder based operator leads to larger and slower circuits (Table 5).

5.3 Feedback Chaining Modes

Table 6 summarizes the characteristics of some IDEA processor suitable for feedback chaining modes. As the rounds are now combinational, the critical path increases and we obtain very low encryption rates. The basic iterative architecture (Figure 7a) seems to be the best one for feedback modes: it requires less slices than systems with partial loop unrolling and achieves the same throughput.

5.4 Comparison with other IDEA Processors

Table 7 digests results published by some other researchers. We give here a brief overview of the main features of these IDEA processors.

1. CryptoBooster (Mosanya *et al.* [13] and Beuchat [1]) is a highly modular and reconfigurable coprocessor taking full advantage of

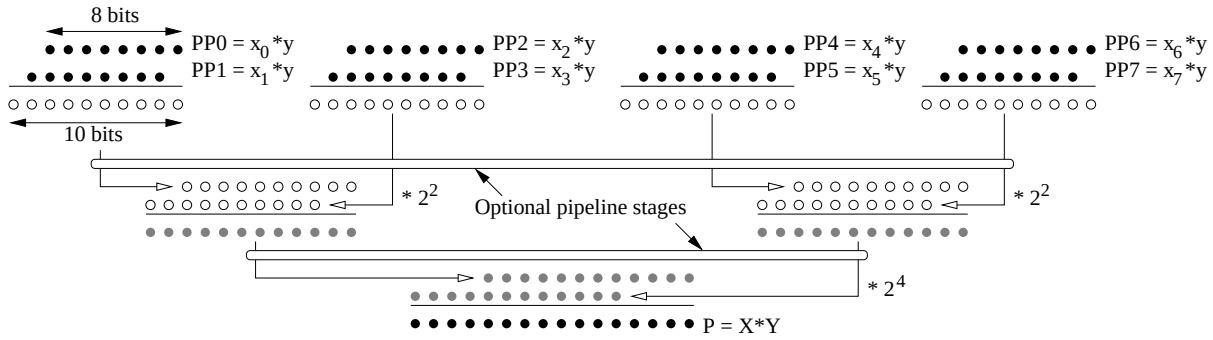

Figure 8: Architecture of an 8×8 unsigned multiplier.

Table 4: Characteristics of our architectures for the three multiplier based $\odot$ operators on Virtex-E devices (no register on boundaries between two successive rounds, $m_1 = 2$, $m_2 = 1$, $m_3 = 1$, $\beta = 1$, and $\gamma = 1$).

Device	Algorithm	Rounds	Slices	Delay [ns]	Through-put [Gb/s]
XCV1600E-6	Low-High	8+1	12959 (83%)	11.7	~ 5.4
XCV1600E-6	Improved Low-High	8+1	12375 (79%)	11.6	~ 5.5
XCV1600E-6	$(n+1) \times (n+1)$ multiplier	8+1	11948 (76%)	18.3	~ 3.5

current FPGAs and working with a host system in order to accelerate cryptographic algorithms. CryptoBooster implements the IDEA block cipher and two chaining algorithms, namely ECB and CBC[2]. Modulo $(2^n + 1)$ is carried out with the Low-High algorithm and the IDEA core consists in a single round and the output transformation (Figure 7a). The main drawback CryptoBooster is the complexity of its control units: the coprocessor is divided into several modules responsible for memory management, communication with the host system, data encryption, or block chaining. All these modules communicate together using unidirectional point-to-point channels. It is therefore possible to change the encryption algorithm or the block chaining unit, without modifying any other module. The main goal of this architecture is to allow partial reconfiguration of the coprocessor. Unfortunately, CryptoBooster is less efficient and much more expensive than an optimized software implementation.

2. CryptoBooster II (Beuchat *et al.* [2]) is an improved version of CryptoBooster. Thanks to a simplified control unit and a better implementation of the modulo $(2^n + 1)$ multiplier, this coprocessor embeds four rounds and the output transformation (Figure 7c) while shortening the critical path. For the same price as CryptoBooster, the throughput is approximately 40 times higher and outperforms the best software solution.

3. Leong *et al.* [9] have proposed a bit-serial architecture enabling the algorithm to be deeply pipelined to achieve a system clock rate of 125 MHz on

[2]This mode slightly differs from the CBC mode described in the literature: in order to accommodate the latency engendered by pipelining, the chaining algorithm disposes of $q > 1$ initialization vectors.

Table 5: Characteristics of architectures involving the carry-save adder based $\odot$ operator (no register on boundaries between two successive rounds, $m_1 = 2$, $m_2 = 1$, $m_3 = 1$, $m_4 = 1$, $\beta = 1$, and $\gamma = 1$).

Device	Algorithm	Rounds	Slices	Delay [ns]	Throughput [Gb/s]
XC2V4000-6	Carry-save adder based architecture	8+1	18537 (80%)	8.2	~ 7.9
XCV2000E-6	Carry-save adder based architecture	8+1	18164 (94%)	13.8	~ 4.6

Table 6: Characteristics of our architectures for feedback modes. ($\alpha = \beta = \gamma = 0$).

Device	Algorithm	Rounds	Slices	Mult. blocks	Delay [ns]	Throughput [Gb/s]
XC2V250-6	Low-High	1+1	1148 (74%)	6	49.4	~ 0.14
XC2V250-6	Improved Low-High	1+1	1049 (68%)	6	49.7	~ 0.14
XC2V250-6	$(n+1) \times (n+1)$ multiplier	1+1	920 (59%)	6	50.5	~ 0.14
XC2V250-6	Low-High	2+1	1463 (95%)	10	92.9	~ 0.14
XC2V250-6	Improved Low-High	2+1	1290 (83%)	10	96.2	~ 0.13
XC2V250-6	$(n+1) \times (n+1)$ multiplier	2+1	1086 (70%)	10	96.0	~ 0.13

a Virtex XCV300-4. Given more resources, this architecture can be scaled up to achieve higher encryption rate. The idea simply consists in instantiating multiple IDEA core. The design of a modulo $(2^n + 1)$ multiplier is also based on the Low-High algorithm.

4. Haenni has studied software implementations of the IDEA block cipher for Itanium and G4 processors [6]. The code was written in assembly language to benefit from the potential of multimedia instructions. As this software performs eight *separate* encryptions in parallel, it has the same throughput for feedback and non-feedback encryption modes.

5. Hämäläinen *et al.* have implemented the IDEA block cipher on a Virtex XCV1000E device [7]. The algorithm proposed by Ma [11] was used to perform the modified modulo $(2^n + 1)$ multiplication. This implementation achieves a throughput of 6.78 Gb/s with a latency of 132 clock cycles.

6 Conclusions

We have investigated four algorithms to carry out the modified modulo $(2^n + 1)$ multiplication for the IDEA block cipher. Since the latest FPGA families embed multiplier blocks, our operators involve small multiplications and are probably not adequate for VLSI or other FPGA devices. A series of experiments should be performed to confirm this hypothesis. This is however a huge work involving the development of new generators. For instance, the small multipliers should be replaced by operators optimized for VLSI (i.e. modified Booth recoding) to obtain a fair comparison.

Several architectures of the IDEA block cipher for Virtex-E and Virtex-II devices have then been described. Our study illustrates that the choice of an algorithm is strongly related to the target technology. For instance, a 8 Gb/s encryption rate requires devices ranging from a XC2V1000 (multiplication with subsequent modulo correction) to a XC2V4000 (carry-save adder based operator).

Table 7: Results of some other researchers.

Reference	Technology	Throughput [Gb/s]	Frequency [MHz]
Mosanya *et al.* [13], Beuchat [1]	XCV1000-4	0.1	13.2
Beuchat *et al.* [2]	XCV1000-4	4.3	66.7
Leong *et al.* [9]	XCV300-4	0.5	125.0
Haenni [6]	Itanium	0.55	733.0
	G4	0.47	450.0
Hämäläinen *et al.* [7]	XCV1000E-6	6.78	105.9
Fastest circuit of this paper	XC2V1000-6	8.5	133.3

7 Acknowledgments

The author would like to thank the "Ministère Français de la Recherche" (grant # 1048 CDR 1 "ACI jeunes chercheurs"), the Swiss National Science Foundation, and the Xilinx University Program for their support. The author is also grateful to Arnaud Tisserand for his careful reading of the manuscript and suggestions for improvement.

References

[1] J.-L. Beuchat. *Etude et conception d'opérateurs arithmétiques optimisés pour circuits programmables.* PhD thesis, Swiss Federal Institute of Technology Lausanne, 2001. Available from http://www.ens-lyon.fr/~jlbeucha.

[2] J.-L. Beuchat, J.-O. Haenni, H. F. Restrepo, C. Teuscher, F. J. Gómez, and E. Sanchez. Approches matérielles et logicielles de l'algorithme de chiffrement IDEA. *Technique et science informatiques*, 21(2):203–224, 2002.

[3] J.-L. Beuchat and A. Tisserand. Small Multiplier-based Multiplication and Division Operators for Virtex-II Devices. 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 513–522. Springer, 2002.

[4] A. V. Curiger, H. Bonnenberg, and H. Kaeslin. Regular VLSI Architectures for Multiplication Modulo (2^n+1). *IEEE Journal of Solid-State Circuits*, 26(7):990–994, 1991.

[5] M. Dworkin. Recommandation for Block Cipher Modes of Operation, 2001. NIST Special Publication 800-38A.

[6] J.-O. Haenni. *Architecture EPIC et jeux d'instructions multimédias pour applications cryptographiques.* PhD thesis, Swiss Federal Institute of Technology Lausanne, 2002. Available from http://www.haenni.info/.

[7] A. Hämäläinen, M. Tommiska, and J. 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.

[8] X. Lai. *On the Design and Security of Block Ciphers.* ETH Series in Information Processing. Hartung–Gorre Verlag Konstanz, 1992.

[9] M. P. Leong, O. Y. H. Cheung, K. H. Tsoi, and P. H. W. Leong. A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA. In B. Hutchings, editor, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 122–131. IEEE Computer Society, 2000.

[10] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption. Available from http://csrc.nist.gov/encryption/modes/proposed-modes/.

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

[12] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[13] E. Mosanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez. CryptoBooster: A Reconfigurable and Modular Cryprographic Coprocessor. In C. K. Koc and C. Paar, editors, *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems, CHES'99, Worcester, MA*, volume 1717 of *Lecture Notes in Computer Science*, pages 246–256. Springer-Verlag, Berlin, Heidelberg, August 12–13 1999.

[14] R. 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, Adelaide, Australia, April 1999.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399