HAL
open science

Some Modular Adders and Multipliers for Field Programmable Gate Arrays

Jean-Luc Beuchat

To cite this version:

Jean-Luc Beuchat. Some Modular Adders and Multipliers for Field Programmable Gate Arrays. [Research Report] LIP RR-2002-37, Laboratoire de l'informatique du parallélisme. 2002, 2+10p. hal-02101822

HAL Id: hal-02101822

https://hal-lara.archives-ouvertes.fr/hal-02101822

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

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

Some Modular Adders and Multipliers for Field Programmable Gate Arrays

Jean-Luc Beuchat

October 2002

Research Report N° 2002-37

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

Some Modular Adders and Multipliers for Field Programmable Gate Arrays

Jean-Luc Beuchat

October 2002

Abstract

This paper is devoted to the study of number representations and algorithms leading to efficient implementations of modular adders and multipliers on recent Field Programmable Arrays. Our hardware operators take advantage of the building blocks available in such devices: carry-propagate adders, memory blocks, and sometimes embedded multipliers. The first part of the paper describes three basic methodologies to carry out a modulo m addition and presents in more details the design of modulo $(2^n \pm 1)$ adders. The major result is a new modulo $(2^n + 1)$ addition algorithm leading to an area-time efficient implementation of this arithmetic operation on FPGAs. The second part describes a modulo m multiplication algorithm involving small multipliers and memory blocks, and modulo $(2^n + 1)$ multipliers based on Ma's algorithm. We also suggest some improvements of this operator in order to perform a multiplication in the group $(\mathbb{Z}^*_{2^n+1}, \cdot)$.

Keywords: Computer arithmetic, modulo m addition, modulo m multiplication, FPGA

Résumé

Cet article est consacré à l'étude de systèmes de représentation des nombres et d'algorithmes conduisant à des implantations efficaces d'additionneurs et de multiplieurs modulo m sur les FPGA actuels. Nos opérateurs matériels tirent parti des briques de base disponibles dans de tels circuits : additionneurs à retenue propagée, blocs de mémoire et petits multiplieurs. La première partie de l'article décrit trois méthodes de conception d'additionneurs modulo m et présente plus particulièrement des additionneurs modulo $(2^n \pm 1)$. Le principal résultat est un nouvel additionneur modulo $(2^n + 1)$ conduisant à un opérateur compact et rapide sur FPGA. La seconde partie propose un algorithme de multiplication modulo m requérant des petits multiplieurs et des blocs de mémoire. Nous présentons également un multiplieur modulo $(2^n + 1)$ basé sur l'algorithme de Ma et suggérons quelques modifications afin d'effectuer la multiplication dans le groupe $(\mathbb{Z}^*_{2^n+1}, \cdot)$.

Mots-clés: Arithmétique des ordinateurs, addition modulo m, multiplication modulo m, FPGA

1 Introduction

Modular arithmetic plays a crucial role in various fields such as residue number system arithmetic or cryptography. Several algorithms for modular addition and multiplication have been proposed and numerous papers describe both theoretical and practical results (see for instance [4, 6, 7]). Those algorithms are generally designed for standard integrated circuits and are based on very low-level basic elements such as NAND or XOR gates. However, recent Field Programmable Gate Arrays (FPGA) embed dedicated carry logic, memory blocks, and sometimes small multipliers. Arithmetic operators taking advantage of these new building blocks could outperform classic architectures. A recent study of unsigned multiplication and division on Virtex-II devices has already shown that embedded multipliers allow significant speed improvements compared to standard solutions only based on Configurable Logic Blocks (CLB) [2].

This paper is devoted to the study of number representations and algorithms leading to efficient implementations of modular adders (Section 2) and multipliers (Section 3) on Virtex-E and Virtex-II devices. Virtex-E and Virtex-II 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 carry logic dedicated to addition and subtraction. Furthermore, Virtex-E FPGAs incorporate large memory blocks organized in columns. Each block is a fully synchronous dual-ported 4096-bit RAM whose data width can be configured (1, 2, 4, 8 or 16 bits). A Virtex-II device embeds many 18-bit $\times$ 18-bit multipliers supporting two independent dynamic data input ports: 18-bit signed or 17-bit unsigned. 18-Kbit true dual-port RAM blocks (called block SelectRAM resources) accepting various data/address aspect ratios are also available. Arithmetic operators dedicated to FPGAs should therefore involve such building blocks.

2 Modular Addition

The modulo m addition of two numbers x and y belonging to $\{0, \ldots, m-1\}$ is defined by

$$(x+y) \bmod m = \begin{cases} x+y & \text{if } x+y < m, \\ x+y-m & \text{if } x+y \geq m, \end{cases} \tag{1}$$

and can be straightforwardly implemented by an adder, a comparator, and a subtracter. The comparison is however expensive, both in terms of area and delay. The algorithms studied in this section allow to get rid of it and lead to more efficient hardware operators. In this paper, $k = \lfloor \log_2 m \rfloor + 1$ denotes the number of bits which are required to encode both inputs and output of a modulo m arithmetic operator. There are basically three methodologies to carry out a modulo m addition [3]:

- **Table Based Operators.** This solution consists in storing in a table the values $(x+y) \bmod m$ for each pair of inputs x and y (Figure 1a). Its main drawback lies in the exponential growth of the required memory size ($k \cdot 2^{2k}$ bits).
- **Hybrid Operators.** Figure 1b describes a modulo m adder involving a standard binary adder followed by a ROM which corrects the sum. This architecture reduces the memory requirements from $k \cdot 2^{2k}$ bits to $k \cdot 2^{k+1}$ bits.
- **Adder Based Operators.** Another way to implement Equation (1) is described by Algorithm 1 and leads to the circuit of Figure 1c. Reference [3] provides for instance a proof a correctness of this method. This architecture requires only two carry-propagate adders and a multiplexer and is therefore well suited for FPGAs.

Example 1 (Modulo 13 addition)
Let us illustrate the behavior of the adder based operator for $m = 13$. We choose $j = 4$ ($2^3 < 13 < 2^4$), consequently $2^j - m = 3$.

- *For $x = 3$ and $y = 4$, we have $s_0 = 7$ and $s_1 = 10$. Since the carry-out signal of both adders is equal to zero, $(x+y) \bmod m = s_0 \bmod 2^4 = 7$.*
- *For $x = 12$ and $y = 1$, we obtain $s_0 = 13$ and $s_1 = 16$. The carry-out signal of the second adder is therefore equal to one and $(x+y) \bmod m = s_1 \bmod 2^4 = 0$.*
- *For $x = 10$ and $y = 7$, $s_0 = 17$ and $s_1 = (s_0 \bmod 2^4) + 3 = 4$. Since the carry-out bit of s_0 is equal to one, $(x+y) \bmod m = s_1 \bmod 2^4 = 4$.*

2.1 Modulo $(2^n \pm 1)$ Addition

Some improvements of the adder based operator previously described are possible for specific values of m. For instance, modulo $(2^n - 1)$ addition, or one's complement addition, is defined by [10]:

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

Algorithm 1 Modulo m addition.

1: Choose j such that $2^{j-1} < m < 2^j$
2: $s_0 \leftarrow x + y$
3: $s_1 \leftarrow (s_0 \bmod 2^j) + 2^j - m$
4: **if** the carry-out bit of s_0 or s_1 is one **then**
5: $\quad (x + y) \bmod m \leftarrow s_1 \bmod 2^j$
6: **else**
7: $\quad (x + y) \bmod m \leftarrow s_0 \bmod 2^j$
8: **end if**

Figure 1d depicts the architecture of the corresponding hardware operator. Due to the condition $x + y + 1 \geq 2^n$, we perform two additions in parallel and select the correct result with a multiplexer. Remember that zero has a double representation in one's complement, namely "$0 \ldots 0$" and "$1 \ldots 1$" (i.e. 0 is congruent to $2^n - 1$ (modulo $2^n - 1$)). If the computation path accommodates the second encoding of zero, Equation (2) can be rewritten as follows:

$$(x + y) \bmod (2^n - 1) = \begin{cases} (x + y + 1) \bmod 2^n & \text{if } x + y \geq 2^n, \\ x + y & \text{if } x + y < 2^n. \end{cases} \tag{3}$$

Note that the carry-out c_{out} from the sum $x + y$ indicates whether the incrementation must be performed. It is still possible to evaluate $x + y$ and $x + y + 1$ in parallel, and to choose the correct result according to c_{out} (Figure 1e). An alternate architecture, illustrated on Figure 1f, simply adds c_{out} to the $x + y$.

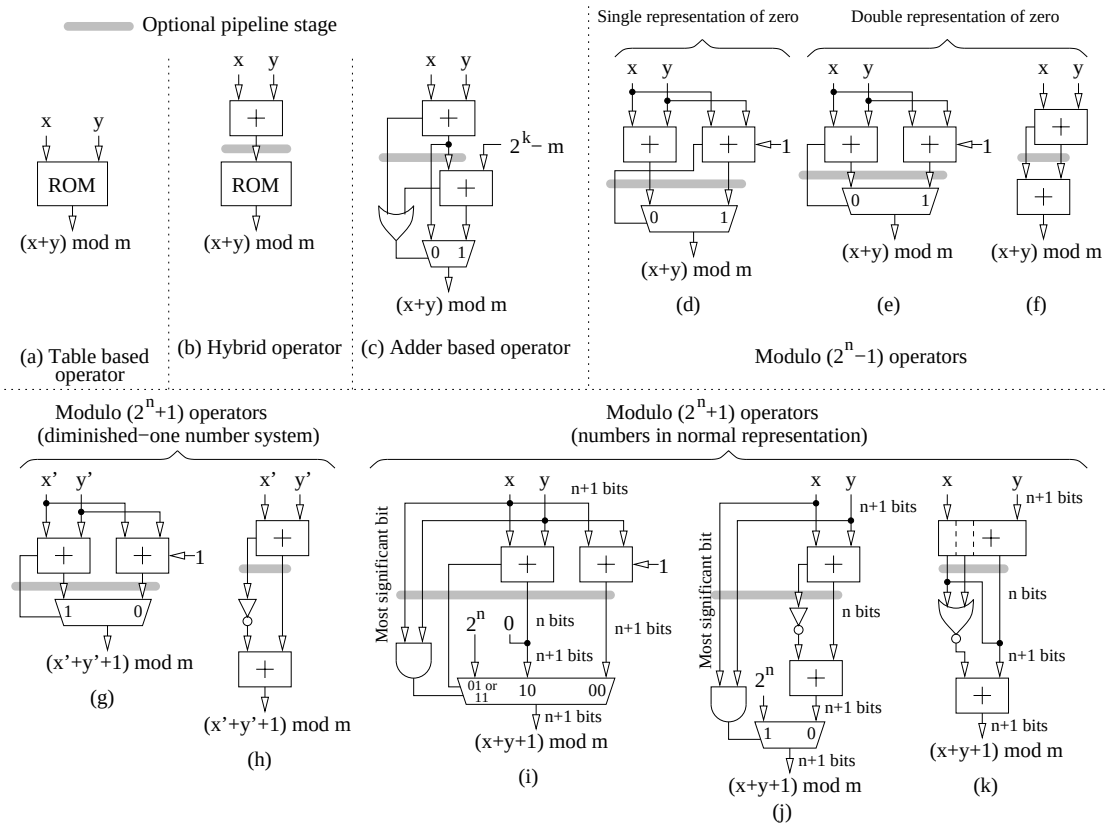

Figure 1: Several architectures of modulo m adders.

Designing a modulo $(2^n + 1)$ adder is a little bit trickier. Such an operator is however useful in a wider range of application including for instance the modulo $(2^n + 1)$ multiplier of the IDEA block cipher [8]. This arithmetic operation is often performed in the diminished-one number system, where a x is represented by $x' = x - 1$ and the number 0 is not used or treated as a special case [10]:

$$\begin{aligned} (x' + y' + 1) \bmod (2^n + 1) &= \begin{cases} x' + y' & \text{if } x' +' y \geq 2^n, \\ (x' + y' + 1) \bmod 2^n & \text{if } x' + y' < 2^n. \end{cases} \\ &= (x' + y' + \bar{c}_{out}) \bmod 2^n. \end{aligned} \tag{4}$$

Figure 1g and Figure 1h depict two hardware operators performing the modulo $(2^n + 1)$ addition according to this algorithm. The principle of these architectures is the same as for the modulo $(2^n - 1)$ adder.

Example 2 (Modulo $(2^8 + 1)$ addition in the diminished-one number system)
Let us describe the behavior of a modulo $(2^8 + 1)$ adder based on Equation (4) for some examples:

- *For $x = 2^8$ and $y = 2^8$, we deduce that $x' = 2^8 - 1$, $y' = 2^8 - 1$, and $(x' + y' + 1) \bmod (2^8 + 1) = 2^8 - 2$. This number is the diminished-one representation of $(x + y) \bmod (2^8 + 1) = 2^8 - 1$.*
- *For $x = 13$ and $y = 4$, we obtain $(x'+y'+1) \bmod (2^8+1) = (12+3+\bar{c}_{out}) \bmod 2^8 = 16$, which is the diminished-one representation of 17.*

Let us study the modulo $(2^n + 1)$ addition of two numbers in normal representation. The algorithms described in this paper returns the desired result increased by one. Nevertheless, this property facilitates the design of the circuit and can be dealt with in many applications. The modulo $(2^n + 1)$ addition is now defined by:

$$\begin{aligned}(x + y + 1) \bmod (2^n + 1) &= \begin{cases} 2^n & \text{if } x = 2^n \text{ and } y = 2^n, \\ x + y + 1 & \text{if } x + y < 2^n, \\ x + y - 2^n & \text{if } 2^n \leq x + y < 2^{n+1} \end{cases} \\ &= \begin{cases} 2^n & \text{if } x = 2^n \text{ and } y = 2^n, \\ (x + y) \bmod 2^n + \bar{c}_{\text{out}} & \text{if } 0 \leq x + y < 2^{n+1}. \end{cases} \end{aligned} \tag{5}$$

Two direct implementations of Equation (5) are illustrated by Figure 1i and Figure 1j [5]. Their main drawback lies in the multiplexer handling the special case where both operands are equal to 2^n.

Example 3 (Modulo $(2^8 + 1)$ addition)
We consider again the modulo $(2^8 + 1)$ addition to show how the operator illustrated by Figure 1j works.

- *For $x = 2^8$ and $y = 2^8$, both inputs of the AND gate are equal to one and the multiplexer selects 2^8. Since $(2^8 + 2^8) \bmod (2^8 + 1) = 2^8 - 1$, the circuit returns the sum increased by one.*
- *For $x = 13$ and $y = 4$, we have $(x + y) \bmod 2^8 = 17$ and $\bar{c}_{out} = 1$. The multiplexer selects now the output of the adder. Consequently, the output is $18 = (13 + 4 + 1) \bmod (2^8 + 1)$.*
- *For $x = 1$ and $y = 2^8 - 2$, we obtain $(x + y) \bmod 2^8 = 2^8 - 1$ and $\bar{c}_{out} = 1$. Since the multiplexer selects again the output of the adder, the result is $(x + y + 1) \bmod (2^8 + 1) = 2^8$.*

We suggest here an alternative architecture suppressing the multiplexer. Let us define the $(n + 2)$-bit integer $s = s_{n+1}s_n \ldots s_0 = x + y$. The modulo $(2^n + 1)$ addition can be expressed as:

$$(x + y + 1) \bmod (2^n + 1) = (x + y) \bmod 2^n + s_{n+1}2^n + \overline{s_{n+1} \vee s_n}. \tag{6}$$

A proof of correctness of this algorithm is provided in Annex A. Figure 1k depicts the resulting hardware operator which requires only two carry-propagate adders and a NOR gate. On Virtex-E or Virtex-II devices, this logic gate is implemented within the carry chain (Figure 2) and the modulo $(2^n + 1)$ adder fits into a single CLB column. Note that this operator also deals with numbers in diminished-one representation, while eliminating the constraint $x, y \neq 0$. The conversion from normal representation to diminished-one number system is now defined by $\xi = (x - 1) \bmod (2^n + 1)$: the number $(0 - 1)$ is congruent to 2^n (modulo $(2^n + 1)$).

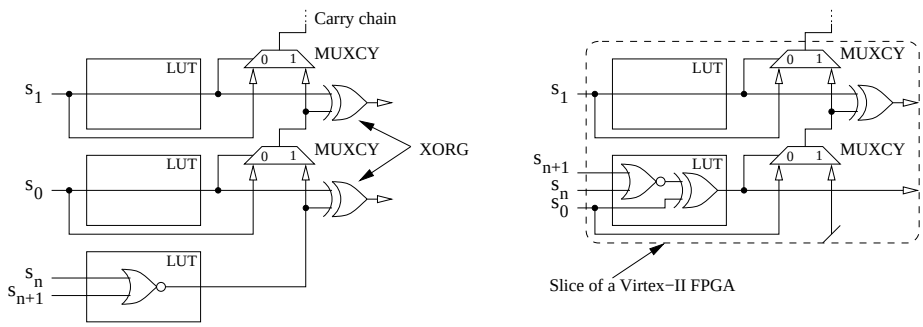

Figure 2: Two implementations of the new modulo $(2^n + 1)$ adder on Virtex-E or Virtex-II devices (detail).

2.2 Implementation Results

We have written a C program which generates the synthesizable VHDL code of each circuit illustrated on Figure 1. Three parameters allow to choose one of the modulo adders, to specify the modulo m, and to insert an optional pipeline stage. We have then conducted a series of experiments with this tool[1] in order to evaluate the area and the delay of each modular adder according to m.

Our first experiment aims to compare four architectures of a modulo (2^n-1) adder (Table 1). The operators depicted by Figure 3d and Figure 3e do not significantly improve the adder based operator defined by Algorithm 1. The last modulo (2^n-1) adder described in this paper (Figure 3)f does not require a multiplexer and is therefore smaller. The delay of the four circuits is comparable. This experiment illustrates that a peculiar number encoding (i.e. the double representation of zero) can sometimes lead to a better hardware implementation of an arithmetic operator.

Table 1: Comparison of some modulo (2^n-1) adders on a XCV50E-6 device.

	$n=4$	$n=8$	$n=12$	$n=16$	$n=20$	$n=24$	$n=28$	$n=32$
Fig. 1c	6 slices 9.4 ns	13 slices 11.5 ns	19 slices 12.3 ns	25 slices 13.4 ns	31 slices 13.7 ns	37 slices 14.3 ns	43 slices 14.5 ns	49 slices 14.9 ns
Fig. 1d	6 slices 8.1 ns	12 slices 9.0 ns	18 slices 9.4 ns	24 slices 10.2 ns	30 slices 10.9 ns	36 slices 11.1 ns	42 slices 12.6 ns	48 slices 13.3 ns
Fig. 1e	6 slices 8.0 ns	12 slices 8.6 ns	18 slices 9.4 ns	24 slices 9.7 ns	30 slices 10.9 ns	36 slices 11.5 ns	42 slices 12.3 ns	48 slices 13.1 ns
Fig. 1f	5 slices 8.4 ns	8 slices 8.8 ns	12 slices 9.4 ns	16 slices 10.7 ns	20 slices 14.8 ns	24 slices 13.7 ns	28 slices 15.5 ns	32 slices 14.2 ns

Table 2 describes the main specificities of some modulo (2^n+1) adders on a Virtex-E device. Due to the required memory size, the hybrid operator (Figure 1b) is rather unattractive and is limited to small moduli ($n \leq 8$). Note that the table based method works only if $n \leq 5$. For $n=4$, the operator requires for instance two 4096-bit RAM blocks and four slices, and its delay is equal to 4.3 ns. For $n=7$, 5219 slices are needed and the delay is then equal to 44.5 ns. This experiment also shows that our new modulo (2^n+1) addition algorithm leads to the smallest circuits.

Table 2: Comparison of some modulo (2^n+1) adders on a XCV50E-6 device.

	$n=4$	$n=8$	$n=12$	$n=16$	$n=20$	$n=24$	$n=28$	$n=32$
Fig. 1b	10 slices 10.2 ns	20 slices 12.5 ns	– –	– –	– –	– –	– –	– –
Fig. 1c	7 slices 9.4 ns	15 slices 10.8 ns	21 slices 12.2 ns	27 slices 13.0 ns	33 slices 13.3 ns	39 slices 13.6 ns	45 slices 14.3 ns	51 slices 16.9 ns
Fig. 1j	6 slices 8.7 ns	13 slices 11.6 ns	19 slices 12.2 ns	25 slices 13.6 ns	31 slices 14.4 ns	37 slices 14.9 ns	43 slices 15.7 ns	49 slices 18.7 ns
Fig. 1k	6 slices 7.8 ns	11 slices 11.6 ns	15 slices 11.6 ns	19 slices 13.4 ns	23 slices 14.5 ns	27 slices 13.8 ns	31 slices 14.7 ns	35 slices 19.2 ns

3 Modular Multiplication

A basic modulo m multiplication algorithm consists in computing $w = xy$, where $0 \leq x, y < m$, and dividing this product by m. Since division is hard to perform, several algorithms have been proposed to overcome this problem. Reference [6] provides a good bibliography on this subject. All these solutions are however dedicated to VLSI implementations; consequently, we propose here a study of some modular multipliers based on the building blocks available in recent FPGAs. Analogously to addition, modulo m multiplication can be implemented by means of tables (Figure 3a). This approach is however limited to small moduli due to the exponential growth of the required memory, and other architectures must be investigated.

3.1 Multiplication with Subsequent Modulo Correction

This modulo m multiplication scheme is dedicated to FPGAs embedding small multipliers and memory blocks. The principle consists in computing the $2k$-bit wide product xy and then performing a modulo correction by means of a

[1] All experiments described in this paper were performed on a Sun Microsystems Ultra-10 workstation (440 MHz, 1 GB of memory). All input and output signals were routed through the D-type flip-flops available in the Input/Output blocks of Virtex-E or Virtex-II devices. The optional pipeline stage has not been used. The automatically generated VHDL code was synthesized using Synplify Pro 7.0.3 and implemented on Virtex-E and Virtex-II devices employing Xilinx Alliance Series 4.1.03i. No specific constraints were given to the synthesis tools and it should be therefore possible to improve the results.

table and a modulo m addition. Given a number 2^j, the algorithm is described as follows:

$$\begin{aligned}(xy) \bmod m &= ((xy) \bmod 2^j + 2^j \cdot (xy) \operatorname{div} 2^j) \bmod m \\ &= \begin{cases} ((xy) \bmod 2^j - (m - 2^j) \cdot (xy) \operatorname{div} 2^j) \bmod m & \text{if } m > 2^j, \\ ((xy) \bmod 2^j + (2^j - m) \cdot (xy) \operatorname{div} 2^j) \bmod m & \text{if } m < 2^j \end{cases} \\ &= \begin{cases} ((xy) \bmod 2^j + (-(m - 2^j) \cdot (xy) \operatorname{div} 2^j) \bmod m) \bmod m & \text{if } m > 2^j, \\ ((xy) \bmod 2^j + ((2^j - m) \cdot (xy) \operatorname{div} 2^j) \bmod m) \bmod m & \text{if } m < 2^j. \end{cases} \end{aligned} \tag{7}$$

The case where $m = 2^j$ is straightforward and will not be addressed in this paper. This scheme requires an unsigned multiplier, a memory to store all possible values of $(-(m - 2^j) \cdot (xy) \operatorname{div} 2^j) \bmod m$ or $((2^j - m) \cdot (xy) \operatorname{div} 2^j) \bmod m$, and a modulo m adder. Let us define $p_0 = (xy) \bmod 2^j$ and $p_1 = (-(m - 2^j) \cdot (xy) \operatorname{div} 2^j) \bmod m$ (or $p_1 = ((2^j - m) \cdot (xy) \operatorname{div} 2^j) \bmod m$). We have now to distinguish the two following cases:

- For $m > 2^j$, $0 \leq p_0, p_1 \leq m - 1$ and we deduce that $0 \leq p_0 + p_1 \leq 2m - 2$. The final addition can therefore be performed with the modulo m adder described in section 2 (Figure 3b).
- For $m < 2^j$, we have $0 \leq p_0 < 2^j$, $0 \leq p_1 < m$, and $0 \leq p_0 + p_1 \leq 2^j + m - 2$. The architecture of the modulo m multiplier depends on $2^j + m - 2$. If this value is strictly smaller than $2m$, the operator is defined by:

$$(xy) \bmod m = \begin{cases} p_0 + p_1 & \text{if } p_0 + p_1 < m, \\ p_0 + p_1 - m & \text{if } m \leq p_0 + p_1 < 2m. \end{cases} \tag{8}$$

  From $2^j + m - 2 < 2m$, we deduce that $2^j \leq m + 1$. Since $m < 2^j$, Equation (8) holds iff $2^j = m + 1$. For other values of 2^j, the modulo m multiplication is formulated as:

$$(xy) \bmod m = \begin{cases} p_0 + p_1 & \text{if } p_0 + p_1 < m, \\ p_0 + p_1 - m & \text{if } m \leq p_0 + p_1 < 2m, \\ p_0 + p_1 - 2m & \text{if } p_0 + p_1 \geq 2m. \end{cases}$$

  Figure 3c illustrates the corresponding hardware operator.

Example 4 (Modulo $(2^n - 1)$ multiplication)
Let us study the design of a modulo $(2^n - 1)$ multiplier according to this algorithm. We choose $j = n$; since $m = 2^n - 1 < 2^n$ we obtain

$$(xy) \bmod (2^n - 1) = ((xy) \bmod 2^n + (xy) \operatorname{div} 2^n) \bmod (2^n - 1).$$

We are in the special case where $m = 2^j + 1$ and the final modular addition can be performed with one of the modulo adder described in Section 2.

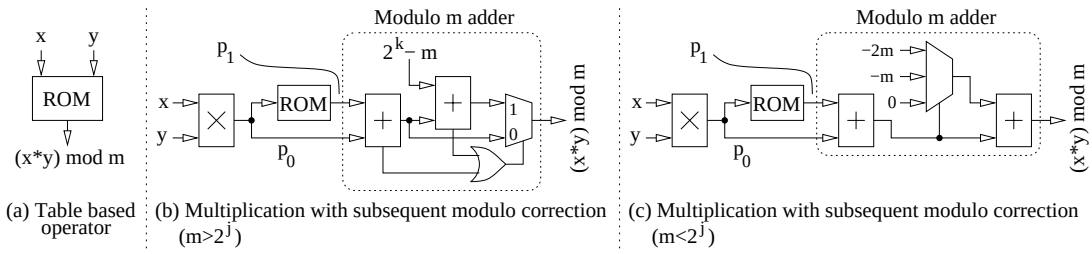

Figure 3: Three architectures of modulo m multipliers.

3.2 Modulo $(2^n + 1)$ Multiplication

Modulo $(2^n + 1)$ multiplication can be performed according to Equation (7). If the target FPGA does not embed small multipliers, the implementation of this scheme is however expensive. We propose an alternate architecture based on the algorithm described by Ma in [9]. Assume that $\psi = \psi_n \ldots \psi_0$ is the diminished-one representation of y, i.e. $\psi = (y - 1) \bmod (2^n + 1)$. When n is even, Ma has proved that

$$xy \bmod (2^n + 1) = \left(\sum_{i=0}^{\frac{n}{2}-1} \underbrace{(2^{2i}(-2\psi_{2i} + \psi_{2i+1} + \psi_{2i-1})x - 1)}_{\mathrm{P}_i} + \frac{n}{2} \right) \bmod (2^n + 1), \tag{9}$$

where $\psi_{-1} = (\bar{\psi}_n \wedge \bar{\psi}_{n-1}) \bmod (2^n+1)$. Each partial product P_i can be easily computed from the diminished-one representation of x (see annex B). When n is odd, the product $xy \bmod (2^n+1)$ is given by

$$xy \bmod (2^n+1) = \Bigg(\overbrace{(2^{n-1}(\psi_{n-1}+\psi_{n-2})x-1)}^{\mathrm{P}_{\frac{n-1}{2}}} + \sum_{i=0}^{\frac{n-1}{2}-1} \underbrace{(2^{2i}(-2\psi_{2i}+\psi_{2i+1}+\psi_{2i-1})x-1)}_{\mathrm{P}_i} + \left(\frac{n-1}{2}+1\right)\Bigg) \bmod (2^n+1), \tag{10}$$

where $\psi_{-1} = \bar{\psi}_n \bmod (2^n+1)$. Ma computes the sum of the partial products and the constant with a carry-save adder, then performs a modulo (2^n+1) reduction with two modulo (2^n+1) carry-save adders and one modulo (2^n+1) carry-propagate adder [9]. This architecture does not take advantage of the fast carry logic available in FPGAs and we suggest here an implementation of Equations (9) and (10) based on our new modulo (2^n+1) adder described in Section 2. Both equations imply to sum up $\lceil n/2 \rceil$ partial products P_i and the constant $\lceil n/2 \rceil$. Remember that our modulo (2^n+1) adder returns the sum of its two operands increased by one. Consequently, if we compute the sum $\sum_{i=0}^{\lceil n/2 \rceil} \mathrm{P}_i$ with $(\lceil n/2 \rceil - 1)$ such adders, we obtain $(\sum_{i=0}^{\lceil n/2 \rceil} \mathrm{P}_i + \lceil n/2 \rceil - 1) \bmod (2^n+1)$, which is the diminished-one representation of the product. Figure 4a depicts the corresponding hardware operator which takes as inputs the diminished-one representations of x and y, and returns $(xy-1) \bmod (2^n+1)$. Since $(x+(2^n-1)+1) \bmod (2^n+1) = (x-1) \bmod (2^n+1)$, conversion from unsigned integer to diminished-one number system can be achieved with our modulo (2^n+1) adder. The inverse conversion is performed with a carry-propagate adder (Figure 4a): it is easy to verify that $(a+1) \bmod (2^n+1) = \sum_{i=0}^{n-1} a_i 2^i + \bar{a}_n$ when $a \in \{0, \ldots, 2^n\}$.

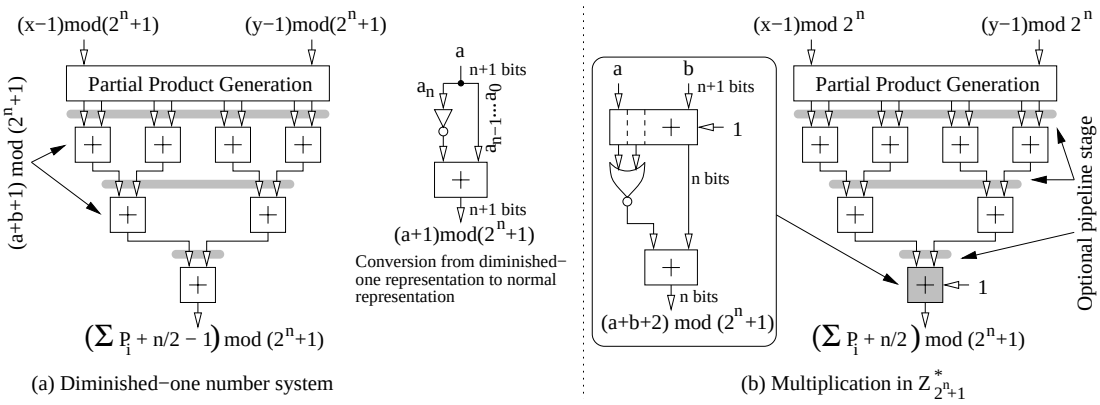

Figure 4: Architectures of a modulo (2^n+1) multiplier based on Ma's algorithm.

Let us consider the multiplication in $\mathbb{Z}^*_{2^n+1} = \{a \in \mathbb{Z}_{2^n+1} \mid \gcd(a, 2^n+1) = 1\}$ (i.e. the multiplicative group of $\mathbb{Z}_{2^n+1}$). Since $(\mathbb{Z}^*_{2^n+1}, \cdot)$ is a group, we know that $(xy) \bmod (2^n+1) \neq 0$ and it is therefore possible to represent the number 2^n by 0. This trick saves one bit and allows two improvements of the multiplier based on Ma's algorithm:

- Due to the special encoding of 2^n, the diminished-one representation of a number $x \in \mathbb{Z}^*_{2^n+1}$ is $(x-1) \bmod 2^n$. We obtain for instance $(0-1) \bmod 2^n = 2^n - 1$, which is the diminished-one representation of 2^n. It is easy to check that $(x-1) \bmod (2^n+1) = (x-1) \bmod 2^n$ when $1 \leq x \leq 2^n - 1$.

- The conversion from diminished-one number system to unsigned integer does not require an additional stage anymore (Figure 4b). It is indeed possible to modify the last adder of the tree in order to compute $(a+b+2) \bmod (2^n+1)$ according to

$$(a+b+2) \bmod (2^n+1) = \left(\sum_{i=0}^{n-1} s_i 2^i + \overline{s_{n+1} \vee s_n} \right) \bmod 2^n, \tag{11}$$

  where $s = s_{n+1} s_n \ldots s_0 = a+b+1$. A proof of correctness of this algorithm is provided in Annex C.

Multiplication in $\mathbb{Z}^*_{2^{16}+1}$ is for instance the critical operation of the IDEA block cipher [8]. Several modulo (2^n+1) multipliers have consequently been investigated over the past years (see for instance [1, 4, 10]). Another implementation of Ma's algorithm has been proposed by Hämäläinen *et al.* in [5]. This architecture is also based on carry-propagate adders. However, modulo (2^n+1) additions are carried out by the circuit of Figure 1j, and an additional stage performs the conversion. This modular multiplier is therefore larger and slower than ours.

Another way to implement Equation (9) or Equation (10) consists in computing the sum $s = \sum_{i=0}^{\lceil n/2 \rceil} \mathrm{P}_i + \lceil n/2 \rceil - 1$ with a carry-propagate adder tree, then in performing a modulo $(2^n + 1)$ correction. We assume that n is even and define $s_0 = s \bmod 2^n$ and $s_1 = s \operatorname{div} 2^n$. Since $0 \leq s_0 \leq 2^n - 1$ and $0 \leq s_1 \leq n/2$, we obtain:

$$\begin{aligned} s \bmod (2^n+1) &= (s_0 + 2^n s_1) \bmod (2^n+1) = (s_0 - s_1) \bmod (2^n+1) \\ &= (s_0 + 2^n - s_1 + 1) \bmod (2^n+1) = (s_0 + \bar{s}_1 + 2) \bmod (2^n+1) \\ &= \begin{cases} s_0 + \bar{s}_1 + 2 & \text{if } s_0 + \bar{s}_1 + 1 < 2^n, \\ s_0 + \bar{s}_1 + 1 - 2^n & \text{if } s_0 + \bar{s}_1 + 1 \geq 2^n, \end{cases} \end{aligned}$$

which is the diminished-one representation of $(xy) \bmod (2^n+1)$. Figure 5a depicts the corresponding hardware operator. Small improvements are again possible when $x, y \in \mathbb{Z}^*_{2^n+1}$. The conversion from normal representation to the diminished-one number system is exactly the same as for the operator based on modulo $(2^n + 1)$ adders (Figure 4b). Note finally that $0 \leq s \bmod (2^n+1) \leq 2^n - 1$. Due to the special encoding of zero, $(xy) \bmod (2^n+1) = (s \bmod (2^n+1) + 1) \bmod 2^n$ and we perform the conversion by setting the input carry of the final adder to one (Figure 5b).

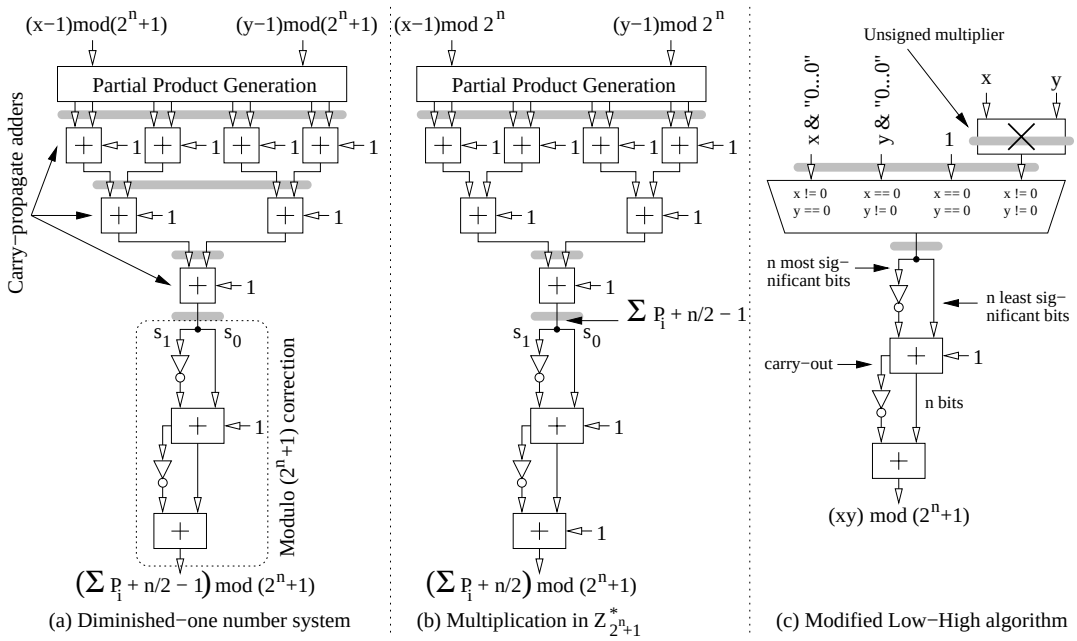

Figure 5: Two other architectures of a modulo $(2^n + 1)$ multiplier based on Ma's algorithm and the architecture based on the modified Low-High algorithm described in [1].

3.3 Implementation Results

This section describes the main characteristics of some modulo m multipliers studied in this paper. The experimental setup is the same as for modular adders. Table 3 digests the main characteristics of modulo m multipliers based on a multiplication with subsequent modulo correction for Virtex-II devices. We only consider here operators requiring a single 18-Kbit memory block, which defines the maximum value for m. Remember that $k = \lfloor \log_2 m \rfloor + 1$ denotes the number of bits required to encode m. When $m \leq 2^k$, Equation (7) yields

$$(xy) \bmod m = ((xy) \bmod 2^k + ((2^k - m) \cdot (xy) \operatorname{div} 2^k) \bmod m.$$

The table is addressed by the k-bit word $(xy) \operatorname{div} 2^k$ and also returns a k-bit number. Consequently, the block SelectRAM is configured in the 1K×18-bit mode (i.e 10 address bits and 18 data bits) and the modulo m is comprised between 3 and 1023.

Table 3: Multiplication with subsequent modulo correction on a XC2V40-6 device. Each operator requires a small multiplier and a single 18-Kbit memory block.

	$m = 5$	$m = 13$	$m = 29$	$m = 61$	$m = 125$	$m = 253$	$m = 509$	$m = 1021$
Area [slices]	6	8	12	13	15	16	20	19
Delay [ns]	5.6	8.5	8.7	9.2	9.5	10.2	9.5	9.7

Table 4 summarizes the area and the delay of several modulo (2^n+1) multipliers when $x, y \in \mathbb{Z}^*_{2^n+1}$. In this experiment, we compare the two architectures discussed in this paper (Figures 4b and 5b) to an operator based on the modified Low-High algorithm proposed in [1]. This circuit involves a $n \times n$ unsigned multiplier, a multiplexer to handle the special cases where $x = 0$ or $y = 0$, and a modulo correction (Figure 5c). Our experiments illustrate that:

- The architecture based on a carry-propagate adder tree and a modulo (2^n+1) correction seems to be the better implementation of Ma's algorithm for FPGAs. This result is not surprising: note that our modulo (2^n+1) adder is roughly twice as large as an $(n+1)$-bit carry-propagate adder. Consequently, the two modulo (2^n+1) multipliers previously described respectively require $2 \cdot \lceil n/2 \rceil$ (Figure 4b) and $\lceil n/2 \rceil + 2$ (Figure 5b) carry-propagate adders to sum the partial products.
- The modified Low-High algorithm leads to smaller and faster circuits when the modulo (2^n+1) multiplier is purely combinatorial. The n-bit $\times$ n-bit unsigned multiplier sums up n partial products $P_i = 2^i x_i y$, $i \in \{0, \ldots, n-1\}$ with a tree of carry-propagate adders. This architecture takes advantage of the dedicated AND gate associated with each LUT in order to generate the partial products. Although Ma's algorithm reduces the amount of partial products, their generation involves much more hardware resources (LUTs and multiplexers).
- However, when pipeline stages are inserted to reduce the delay, the circuit illustrated on Figure 4b is attractive for $n \leq 28$. This result is explained by the fact that pipelining the unsigned multiplier of the operator based on the modified Low-High algorithm is expensive.

Table 4: Modulo (2^n+1) multiplication in $\mathbb{Z}^*_{2^n+1}$ on a XCV200E-6 device.

	$n=4$	$n=8$	$n=12$	$n=16$	$n=20$	$n=24$	$n=28$	$n=32$
Figure 4b (without pipeline)	14 slices 11.6 ns	61 slices 21.5 ns	141 slices 30.9 ns	254 slices 35.5 ns	398 slices 47.3 ns	575 slices 51.8 ns	885 slices 56.6 ns	1156 slices 55.8 ns
Figure 4b (with pipeline)	15 slices 6.7 ns	66 slices 8.9 ns	156 slices 9.7 ns	264 slices 10.8 ns	433 slices 13.3 ns	603 slices 12.4 ns	1014 slices 15.1 ns	1300 slices 15.1 ns
Figure 5b (without pipeline)	15 slices 11.2 ns	57 slices 18.8 ns	123 slices 24.1 ns	212 slices 28.7 ns	325 slices 34.8 ns	461 slices 35.9 ns	725 slices 39.0 ns	939 slices 42.6 ns
Figure 5b (with pipeline)	18 slices 5.8 ns	63 slices 8.5 ns	138 slices 9.9 ns	223 slices 10.6 ns	362 slices 10.9 ns	494 slices 11.5 ns	856 slices 12.1 ns	1089 slices 13.2 ns
Figure 5c (without pipeline)	19 slices 16.1 ns	53 slices 21.8 ns	111 slices 27.6 ns	182 slices 30.0 ns	270 slices 34.5 ns	372 slices 37.9 ns	492 slices 39.3 ns	629 slices 44.2 ns
Figure 5c (with pipeline)	25 slices 7.1 ns	77 slices 9.4 ns	139 slices 11.4 ns	220 slices 12.6 ns	353 slices 12.8 ns	460 slices 14.1 ns	592 slices 13.5 ns	743 slices 15.9 ns

4 Conclusion

We have described and compared several modular adders and multipliers involving various building blocks (carry-propagate adders, tables, and small multipliers). Our main results include the design of a new modulo (2^n+1) adder, a modulo m multiplier based on the embedded multipliers and memory blocks available in Virtex-II devices, and implementations of Ma's algorithm carefully optimized for FPGAs. Our experiments indicate that the choice of an operator depends on several parameters such as the modulo m, the target FPGA family, and the number of internal pipeline stages. However, our VHDL generators allow to quickly explore a wide parameter space and to determine which architecture is most appropriate for a given application.

5 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.

A Proof of the New Modulo (2^n+1) Addition Algorithm

Let us demonstrate that Equation (6) carries out $(x+y+1) \bmod (2^n+1)$ when $0 \leq x, y \leq 2^n$. First of all, let us note that $0 \leq x+y \leq 2^{n+1}$ and

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

We have to distinguish the three following cases to establish the correctness of our algorithm:

- For $x + y = 2^{n+1}$ (i.e. $x = y = 2^n$), we have $(x + y) \bmod 2^n = 0$, $s_n = 0$, and $s_{n+1} = 1$. Our algorithm returns $(x + y + 1) \bmod (2^n + 1) = 2^n = x + y - 2^n$, which is the correct result.
- For $2^n \leq x+y < 2^{n+1}$, we know that $s_{n+1} = 0$ and $s_n = 1$. Consequently, $(x+y+1) \bmod (2^n+1) = (x+y) \bmod 2^n = x + y - 2^n$.
- Finally, for $0 \leq x+y < 2^n$, $s_{n+1} = s_n = 0$ and $(x+y) \bmod 2^n = x+y$. We obtain $(x+y+1) \bmod (2^n+1) = x+y+1$.

B Modulo Partial Product Generation for Ma's Algorithm

Note that $-2\psi_{2i}+\psi_{2i+1}+\psi_{2i-1} \in \{-2, -1, 0, 1, 2\}$. Each partial product has therefore the form $(\pm 2^j x - 1) \bmod (2^n + 1)$. Let $\xi = (x - 1) \bmod (2^n + 1)$ denote the diminished-one representation of x. We describe here how to compute a partial product from ξ. Let us consider the case where $x \neq 0$ (i.e. $\xi \neq 2^n$). Since 2^{n+q} is congruent to -2^q (modulo $(2^n + 1)$), we have:

$$\begin{aligned}
(2^j x - 1) \bmod (2^n + 1) &= (2^j \xi + 2^j - 1) \bmod (2^n + 1) \\
&= \left(\sum_{i=0}^{n-1-j} \xi_i 2^{i+j} + \sum_{i=n-j}^{n-1} \xi_i 2^{i+j} + 2^j - 1 \right) \bmod (2^n + 1) \\
&= \left(\sum_{i=0}^{n-1-j} \xi_i 2^{i+j} - \sum_{i=n-j}^{n-1} \xi_i 2^{i+j-n} + 2^j - 1 \right) \bmod (2^n + 1) \\
&= \left(\sum_{i=0}^{n-1-j} \xi_i 2^{i+j} + \sum_{i=n-j}^{n-1} \bar{\xi}_i 2^{i+j-n} \right) \bmod (2^n + 1), \qquad (12)
\end{aligned}$$

and

$$\begin{aligned}
(-2^j x - 1) \bmod (2^n + 1) &= (-2^j \xi - 2^j - 1) \bmod (2^n + 1) \\
&= \left(- \sum_{i=0}^{n-1-j} \xi_i 2^{i+j} - \sum_{i=n-j}^{n-1} \xi_i 2^{i+j} - 2^j - 1 \right) \bmod (2^n + 1) \\
&= \left(- \sum_{i=0}^{n-1-j} \xi_i 2^{i+j} + \sum_{i=n-j}^{n-1} \xi_i 2^{i+j-n} + 2^n - 2^j \right) \bmod (2^n + 1) \\
&= \left(\sum_{i=0}^{n-1-j} \bar{\xi}_i 2^{i+j} + \sum_{i=n-j}^{n-1} \xi_i 2^{i+j-n} \right) \bmod (2^n + 1). \qquad (13)
\end{aligned}$$

When the modulo $(2^n + 1)$ multiplication is performed in $\mathbb{Z}^*_{2^n+1}$, all partial products are generated according to Equation (12) and Equation (13). However, in the general case, we must also handle $x = 0$ and obtain

$$(2^j x - 1) \bmod (2^n + 1) = (2^j \cdot 2^n + 2^j - 1) \bmod (2^n + 1) = (-2^j + 2^j - 1) \bmod (2^n + 1) = 2^n,$$

and

$$(-2^j x - 1) \bmod (2^n + 1) = (-2^j \cdot 2^n - 2^j - 1) \bmod (2^n + 1) = (2^j - 2^j - 1) \bmod (2^n + 1) = 2^n.$$

C Proof of Equation (11)

The modified modulo $(2^n + 1)$ adder defined by Equation (11) outputs the product p of two elements $x, y \in \mathbb{Z}^*_{2^n+1}$ under the operation of multiplication modulo $(2^n + 1)$. Since $(\mathbb{Z}^*_{2^n+1}, \cdot)$ is a group, we know that $p \in \{a \in \mathbb{Z}_{2^n+1} \mid \gcd(a, 2^n + 1) = 1\}$ (thus $p \neq 0$ and $p \neq 2^n + 1$). Therefore, the sum $a + b + 2$ is not a multiple of $2^n + 1$ and

$$(a + b + 2) \bmod (2^n + 1) = \begin{cases} a + b + 2 & \text{if } 1 < a + b + 1 \leq 2^n - 2, \\ 0 & \text{if } a + b + 1 = 2^n - 1 \text{ or } a + b + 1 = 2^{n+1} \quad (0 \text{ encodes } 2^n), \\ a + b + 1 - 2^n & \text{if } 2^n < a + b + 1 < 2^{n+1}. \end{cases}$$

It is easy to show that Equation (11) returns the correct result for each case described above. Remember that we have defined $s = s_{n+1} s_n \ldots s_0$ such that $s = a + b + 1$.

- For $0 < s = a + b + 1 \leq 2^n - 2$, we obtain $s_{n+1} = s_n = 0$ and $(\sum_{i=0}^{n-1} s_i 2^i + \overline{s_{n+1} \vee s_n}) \bmod 2^n = \sum_{i=0}^{n-1} s_i 2^i + 1 = a + b + 2$.

- For $s = a + b + 1 = 2^n - 1$, we have again $s_{n+1} = s_n = 0$ and $(\sum_{i=0}^{n-1} s_i 2^i + \overline{s_{n+1} \vee s_n}) \bmod 2^n = 2^n \bmod 2^n = 0$. When $s = a + b + 1 = 2^{n+1}$, the most significant bit s_{n+1} is equal to 1 and $s_i = 0$, $\forall i \in \{0, \ldots, n\}$. The operator returns the number 0 which encodes 2^n.
- Finally, for $2^n < s = a+b+1 < 2^{n+1}$, $s_{n+1} = 0$, $s_n = 1$, and $(\sum_{i=0}^{n-1} s_i 2^i + \overline{s_{n+1} \vee s_n}) \bmod 2^n = (\sum_{i=0}^{n-1} s_i 2^i) \bmod 2^n = s - 2^n$, which is the correct result.

References

[1] J.-L. Beuchat. Modular Multiplication for FPGA Implementation of the IDEA Block Cipher. In *Proceedings of the Eleventh ACM International Symposium on Field-Programmable Gate Arrays*, 2003. Submitted.

[2] 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.

[3] A. V. Curiger. *VLSI Architectures for Computations in Finite Rings and Fields*, volume 26 of *Series in Microelectronics*. Hartung-Gorre Verlag, 1993.

[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] 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.

[6] A. A. Hiasat. New Efficient Structures for a Modular Multiplier for RNS. *IEEE Transactions on Computers*, 49(2):170–174, 2000.

[7] A. A. Hiasat. High-Speed and Reduced-Area Modular Adder Structures for RNS. *IEEE Transactions on Computers*, 51(1):84–89, 2002.

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

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

[10] 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.