Automatic Generation of Modular Multipliers for FPGA Applications

Jean-Luc Beuchat and Jean-Michel Muller, *Senior Member, IEEE*

LIP Research Report N° 2007–1
LIP/Arénaire, CNRS – INRIA – ENS Lyon – Université Lyon 1

***Abstract*— Since redundant number systems allow for constant time addition, they are often at the heart of modular multipliers designed for public key cryptography (PKC) applications. Indeed, PKC involves large operands (160 to 1024 bits) and several researchers proposed carry-save or borrow-save algorithms. However, these number systems do not take advantage of the dedicated carry logic available in modern Field-Programmable Gate Arrays (FPGAs). To overcome this problem, we suggest to perform modular multiplication in a high-radix carry-save number system, where a *sum bit* of the carry-save representation is replaced by a *sum word*. Two digits are then added by means of a small Carry-Ripple Adder (CRA). Furthermore, we propose an algorithm which selects the best high-radix carry-save representation for a given modulus, and generates a synthesizable VHDL description of the operator.**

***Index Terms*— Modular multiplication, high-radix carry-save number system, FPGA.**

I. Introduction

THIS paper is devoted to the study of modular multiplication of large operands on Field-Programmable Gate Arrays (FPGAs). This operation is crucial in many public key cryptosystems (*e.g.* elliptic curve cryptography, XTR, RSA) and various solutions have already been investigated. Since iterative algorithms offer a good trade-off between calculation time and circuit area, they have received considerable attention. Least-significant-digit-first schemes are often based on Montgomery's algorithm [1]. However, that approach requires pre- and post-processing and is of interest when a large amount of consecutive modular multiplications is required (*e.g.* modular exponentiation). In this paper, we will consider a most-significant-digit-first scheme.

A. Horner's Rule-Based Modular Multiplication

In order to compute $\langle XY \rangle_M = XY \bmod M$, where M is an n-bit integer such that $2^{n-1} < M < 2^n$, our algorithm is described by an iterative procedure based on the celebrated Horner's rule:

$$\langle XY \rangle_M = \langle (\ldots((x_{r-1}Y)2 + x_{r-2}Y)2 + \ldots)2 + x_0 Y \rangle_M ,$$

where $X = x_{r-1}x_{r-2}\ldots x_1 x_0$ is an unsigned r-bit integer and Y is an n-bit integer belonging to $\{0, \ldots, M-1\}$. This equation can be expressed recursively as follows (we perform a modulo M reduction at each step in order to keep an n-bit intermediate result):

$$\begin{aligned} T[i] &= 2Q[i+1] + x_i Y, \\ Q[i] &= \langle T[i] \rangle_M , \end{aligned} \quad (1)$$

where $Q[r] = 0$ and $Q[0] = \langle XY \rangle_M$. Since $Q[i+1]$ and Y are less than or equal to $M-1$, $T[i] < 3M$ and Equation (1) is implemented by means of a left shift, an addition, a comparator, and up to two subtractions to perform the modulo M reduction [2].

Since public key cryptography involves large integers, operands are often represented in the carry-save number system, which enables addition in constant time (see for instance [3]). However, due to the redundancy of this representation, comparison requires a conversion in a non-redundant number system. This operation involves carry propagations, thus losing the main advantage of the carry-save representation. Several improvements of the algorithm sketched by Equation (1) have therefore been investigated to avoid comparisons. They are based on the following observation: computing a number $P[i]$ congruent to $Q[i]$ modulo M only requires to inspect a few most significant digits of $T[i]$. In order to avoid an expensive final modular reduction, $P[i]$ should be less than a very small multiple of M. The iteration described in this paper returns for instance $\langle XY \rangle_M$ or $\langle XY \rangle_M + M$, and requires at most one subtraction to get the final result.

Koç and Hung [4], [5] proposed for instance a carry-save algorithm based on a sign estimation technique. At each step $-M$, 0, or M is added to $T[i]$ according to a few most significant digits of $Q[i+1]$. Takagi and Yajima [6], [7] applied a similar technique to design signed-digit-based architectures. When the modulus M is known at design time, which is often the case in public key cryptography, another approach consists in building a table $\psi(a) = \left\langle a \cdot 2^{\beta} \right\rangle_M$ and in defining the following iteration:

$$T[i] = 2P[i+1] + x_i Y, \quad (2)$$

$$P[i] = \psi(T[i] \text{ div } 2^{\beta}) + \langle T[i] \rangle_{2^{\beta}} , \quad (3)$$

where $P[r] = 0$ and β is generally chosen equal to n or $n-1$. Since $\psi(T[i] \text{ div } 2^{\beta})$ is an n-bit number, $P[i]$ and $T[i]$ are respectively $(n+1)$- and $(n+3)$-bit numbers. Therefore, the algorithm sketched by the above equations requires a small table. Carry-save implementations of Equations (2) and (3) have for example been proposed by Jeong and Burleson [8], Kim and Sobelman [9], and Peeters *et al.* [10]. Since these algorithms depend on the modulus M, they seem likely candidates for cryptographic hardware based on FPGAs: the reconfigurability of these devices allows one to optimize the architecture according to some parameters (*e.g.* the modulus) and to modify the hardware whenever they change.

B. FPGA-Specific Issues

Modern FPGAs are mainly designed for digital signal processing applications involving rather small operands (16 to 32

Jean-Luc Beuchat is with the University of Tsukuba, Japan. J.-M. Muller is with the *CNRS, laboratoire LIP, projet Arénaire*, France.

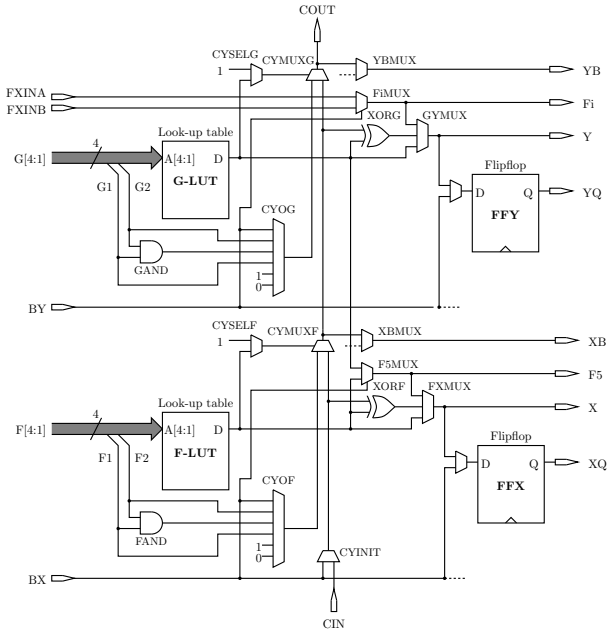


Fig. 1. Simplified diagram of a slice of a Spartan-3 FPGA.

bits). FPGA manufacturers chose to include dedicated carry logic enabling the implementation of fast carry-ripple adders (CRA) for such operand sizes. Let us study for example the architecture of a Spartan-3 device. Figure 1 describes the simplified architecture of a slice, which is the main logic resource for implementing synchronous and combinatorial circuits. Each slice embeds two 4-input function generators (G-LUT and F-LUT), two storage elements (*i.e.* flipflops FFY and FFX), carry logic (CYSELG, CYMUXG, CYSELF, CYMUXF, and CYINIT), arithmetic gates (GAND, FAND, XORG, and XORF), and wide-function multiplexers. Each function generator is implemented by means of a programmable look-up table (LUT). Recall that a full-adder (FA) cell has two bits x_i and y_i, as well as a carry-in bit c_{in} as inputs, and computes a sum bit s_i and a carry-out bit c_{out} such that $2c_{\text{out}} + s_i = x_i + y_i + c_{\text{in}}$. Let $h_i = x_i \oplus y_i$. Then, we have:

$$s_i = h_i \oplus c_{\text{in}}, \tag{4}$$

$$c_{\text{out}} = \begin{cases} x_i & \text{if } h_i = 0 \text{ (\textit{i.e.} } x_i = y_i), \\ c_{\text{in}} & \text{otherwise.} \end{cases} \tag{5}$$

Assume that the F-LUT function generator computes h_i. Then, the XORF gate implements Equation (4), whereas Equation (5) involves three multiplexers (CYOF, CYSELF, and CYMUXF). s_i is either sent to another slice (output X) or stored in a flipflop (FFX). The G-LUT function generator allows one to implement a second FA cell within the same slice, which thus embeds a 2-bit CRA. Unfortunately, a conventional carry-save adder (CSA) requires twice as much hardware resources: since each slice has a single input carry CIN, it is impossible to implement two FA cells with *independent* carry-in bits. Therefore, hardware design tools allocate two function generators when they are provided with a VHDL description of Equations (4) and (5). It is of course possible to write a VHDL code which explicitly instantiates F-LUT, XORF, and CYMUXF. In this case, note that G-LUT can only be used to implement the control unit or the $\psi(.)$ table of Equation (3). According to experiment results, G-LUT often remains unused. Though reducing the number of LUTs of the design, taking advantage of dedicated logic to describe a CSA leads to a larger operator (Table I). Similar problems arise for instance on all Virtex devices (Xilinx) and Cyclone II FPGAs (Altera). It is therefore interesting to investigate modular multiplication algorithms based on FPGA-friendly number systems.

TABLE I
AREA AND NUMBER OF LUTS OF THREE CARRY-SAVE ITERATION STAGES (SPARTAN-3 FPGA).

Algorithm	Without carry logic		With carry logic	
	$n = 32$	$n = 64$	$n = 32$	$n = 64$
Jeong and Burleson [8]	107 slices 200 LUTs	210 slices 392 LUTs	119 slices 141 LUTs	232 slices 271 LUTs
Kim and Sobelman [9]	74 slices 139 LUTs	166 slices 268 LUTs	93 slices 123 LUTS	188 slices 249 LUTs
Peeters *et al.* [10]	74 slices 140 LUTs	160 slices 271 LUTs	95 slices 95 LUTs	190 slices 190 LUTs

C. Our Contribution

We proposed a family of radix-2 algorithms designed for FPGAs embedding 4-input LUTs and dedicated carry logic in [11]. Table II compares our iteration stage against three carry-save schemes. Since these results do not include the conversion from carry-save to unsigned integer which occurs at the end of each multiplication, both area and delay of carry-save operators are underestimated. According to this experiment, our approach is efficient for moduli up to 32 bits. Thus, the aim of this paper is to extend our work to larger moduli. In order to benefit from dedicated carry logic available in almost all FPGA families, we suggest to choose a high-radix carry-save number system, where each *sum bit* of the carry-save representation is replaced by a *sum word* (Section II). Such a number system allows for the design of a modular multiplication algorithm based on small CRAs (Section III). The main originality of our approach is to analyze the modulus M in order to select the most efficient high-radix carry-save representation and to automatically generate the VHDL description of the operator (Section IV). Experimental results validate the efficiency of the proposed modular multiplication scheme (Section V). We proposed a preliminary version of this work based on a different iteration in [12].

TABLE II
AREA AND DELAY OF CARRY-SAVE AND RADIX-2 ITERATION STAGES (SPARTAN-3 FPGA).

Algorithm	$n = 16$	$n = 32$	$n = 64$
Jeong and Burleson [8]	58 slices 9 ns	127 slices 11 ns	236 slices 14 ns
Kim and Sobelman [9]	41 slices 8 ns	79 slices 10 ns	150 slices 12 ns
Peeters *et al.* [10]	50 slices 8 ns	86 slices 11 ns	163 slices 12 ns
Beuchat and Muller [11]	21 slices 12 ns	40 slices 14 ns	80 slices 20 ns

II. HIGH-RADIX CARRY-SAVE NUMBERS

A k-digit high-radix carry-save number X is denoted by

$$X = (x_{k-1}, \ldots, x_0) = \left(\left(x_{k-1}^{(c)}, x_{k-1}^{(s)}\right), \ldots, \left(x_0^{(c)}, x_0^{(s)}\right)\right),$$

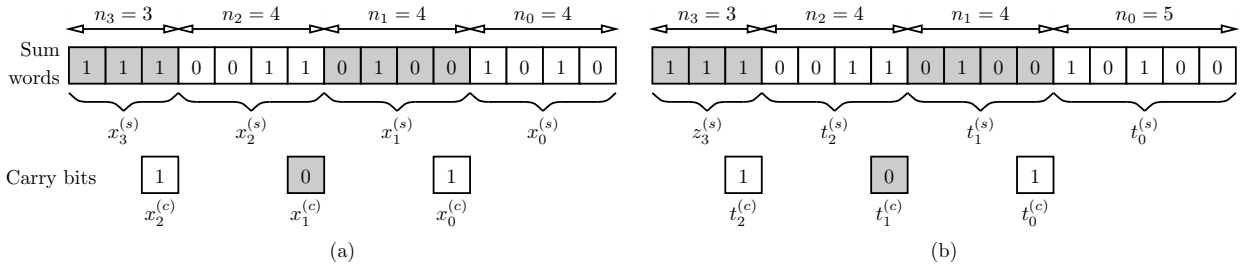


Fig. 2. High-radix carry-save numbers. (a) Encoding of the number $X = 33626$. (b) Encoding of $Z = 2X$.

where the jth digit x_j consists of an n_j-bit sum word $x_j^{(s)}$ and a carry bit $x_j^{(c)}$ such that $x_j = x_j^{(s)} + x_j^{(c)} 2^{n_j}$. According to this definition, we have:

$$\begin{aligned} X &= x_0 + x_1 2^{n_0} + x_2 2^{n_0+n_1} + \ldots + x_{k-1} 2^{n_0+\ldots+n_{k-2}} \\ &= x_0^{(s)} + \sum_{i=0}^{k-2} \left(x_i^{(c)} + x_{i+1}^{(s)} \right) 2^{\sum_{j=0}^{i} n_i} + x_{k-1}^{(c)} 2^{\sum_{j=0}^{k-1} n_i}. \end{aligned}$$

Let us define

$$X^{(s)} = x_0^{(s)} + x_1^{(s)} 2^{n_0} + \ldots + x_{k-1}^{(s)} 2^{n_0+\ldots+n_{k-2}},$$

and

$$X^{(c)} = x_0^{(c)} 2^{n_0} + x_1^{(c)} 2^{n_0+n_1} + \ldots + x_{k-1}^{(c)} 2^{n_0+\ldots+n_{k-1}}.$$

With this notation, we have $X = X^{(s)} + X^{(c)}$. Such a number system has nice properties to deal with large numbers on FPGA:

- its redundancy allows one to perform addition in constant time (the critical path only depends on $\max_{0 \le j \le k-1} n_j$);
- the addition of a sum word $x_j^{(s)}$, a carry bit $x_{j-1}^{(c)}$, and an n_j-bit unsigned binary number can be performed by a CRA.

A key observation is that all sum words do not need to have the same width. This peculiarity will allow us to select a number system according to the modulus to optimize our operators (Section IV). In the following, we will assume that the carry bit of the most significant digit is always equal to zero (the weight of the most significant carry bit is therefore equal to $2^{n_0+n_1+\ldots+n_{k-2}}$).

Example 1 *Figure 2a describes a 4-digit high-radix carry-save number with $n_0 = n_1 = n_2 = 4$ and $n_3 = 3$. According to the first definition of this number system, we have:*

$$\begin{aligned} X &= x_0^{(s)} + \left(x_0^{(c)} + x_1^{(s)} \right) \cdot 2^4 + \left(x_1^{(c)} + x_2^{(s)} \right) \cdot 2^8 + \\ &\quad \left(x_2^{(c)} + x_3^{(s)} \right) \cdot 2^{12} \\ &= 10 + (1+4) \cdot 2^4 + (0+3) \cdot 2^8 + (1+7) \cdot 2^{12} \\ &= 33626. \end{aligned}$$

We can also compute

$$X^{(s)} = 10 + 4 \cdot 2^4 + 3 \cdot 2^8 + 7 \cdot 2^{12} = 29514$$

and

$$X^{(c)} = 1 \cdot 2^4 + 0 \cdot 2^8 + 1 \cdot 2^{12} = 4112.$$

We obtain $X = X^{(s)} + X^{(c)} = 33626$.

Consider the modular multiplication described by Equations (2) and (3) and assume that both $T[i]$ and $P[i]$ are high-radix carry-save numbers. Each equation involves now the addition of a high-radix carry-save number and an unsigned integer (a partial product $x_i Y$ or a number $\psi(T[i] \text{ div } 2^\beta)$ stored in the table). Figure 3 describes how to perform these operations: the integer operand is split into k words of respective lengths $n_0, \ldots, n_{k-1}$; then, each of these words is added to a sum word and a carry bit by means of an n_j-bit CRA. The high-radix carry-save encoding has unfortunately a drawback in the sense that shifting an operand modifies its representation. The following example illustrates this problem, which occurs in the computation of $T[i]$ (Equation (2)).

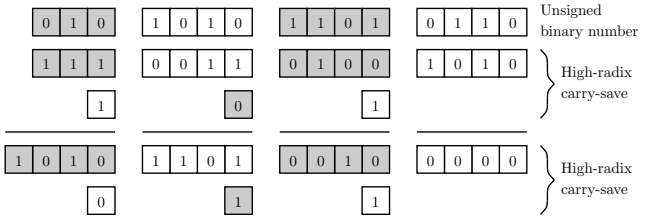


Fig. 3. Addition of an unsigned binary number and a high-radix carry-save number.

Example 2 *Let us consider again the number $X = 33626$, whose format is defined in Figure 2a. By shifting X, we obtain $Z = 2X = 67252$ (Figure 2b). However, the least significant sum word is now a 5-bit number and*

$$\begin{aligned} Z &= z_0 + z_1 2^{n_0+1} + z_2 2^{n_0+n_1+1} + z_3 2^{n_0+n_1+n_2+1} \\ &= 20 + (1+4) \cdot 2^5 + (0+3) \cdot 2^9 + (1+7) \cdot 2^{13} \\ &= 67252. \end{aligned}$$

According to this example, $P[i+1]$ and $P[i]$ do not have the same encoding, and the width of the CRA dealing with the least significant digit of P would increase at each step. The solution consists in converting $T[i]$ to the format of $P[i]$. In the following, we describe a modular multiplication algorithm which minimizes the hardware overhead introduced by this conversion.

III. High-Radix Carry-Save Modular Multiplication

This section describes how to take advantage of a high-radix carry-save number system to perform a modular multiplication. We assume that:

- The modulus M is an n-bit number belonging to $\{2^{n-1} + 1, \ldots, 2^n - 1\}$.
- X is an r-bit unsigned integer.
- Y is an unsigned integer smaller than M.
- α is a small integer parameter which will determine the size of the table required for the modulo M reduction.

- The most significant sum word of $P[i]$ contains at least $n_{k-1} = 5$ bits if $\alpha = 1$ and $n_{k-1} = 6$ bits if $\alpha = 2$. These hypotheses guarantee that
  - $P^{(c)}[i]$ is smaller than $2^{n-\alpha}$, *i.e.* $\left\langle P^{(c)}[i] \right\rangle_{2^{n-\alpha}} = P^{(c)}[i]$ (we fixed the carry bit of the most significant digit of a high-radix carry-save number to zero in Section II), and
  - $P^{(s)}[i]$ is an $(n+2)$-bit number (a proof is given in Appendix)
- At each iteration, we compute a high-radix carry-save number $P[i]$ congruent to $2P[i+1] + x_i Y$ modulo M.

According to our hypotheses, we have:

$$\begin{aligned} P[i+1] &= \left(P^{(s)}[i+1] \text{ div } 2^{n-\alpha}\right) \cdot 2^{n-\alpha} \\ &\quad + \left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}} + \left\langle P^{(c)}[i+1] \right\rangle_{2^{n-\alpha}} \\ &= \left(P^{(s)}[i+1] \text{ div } 2^{n-\alpha}\right) \cdot 2^{n-\alpha} \\ &\quad + \left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}} + P^{(c)}[i+1]. \end{aligned}$$

The iteration of our algorithm is slightly different from the one described in Section I. Let us define $\rho = P^{(s)}[i+1] \text{ div } 2^{n-\alpha}$ and write the intermediate result at step $i+1$ as follows:

$$P[i+1] = \rho 2^{n-\alpha} + \left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}} + P^{(c)}[i+1].$$

It is worth noticing that, according to our hypotheses, ρ is a 3- or 4-bit unsigned number for $\alpha = 1$ or $\alpha = 2$ respectively. Thus, a small table addressed by ρ allows one to efficiently compute a number congruent to $P[i+1]$ modulo M:

$$\begin{aligned} P[i+1] &\equiv \left\langle \rho 2^{n-\alpha} \right\rangle_M + \left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}} \\ &\quad + P^{(c)}[i+1] \pmod{M}. \end{aligned}$$

Note that, when $\alpha = 1$, the table can be stored within the LUTs of a CRA on Spartan-3 and Virtex FPGAs [11]. Since we compute a high-radix carry-save number congruent to XY modulo M, a conversion and a final modulo M reduction are mandatory. In order to keep the hardware overhead as small as possible, we apply a trick proposed by Peeters *et al.* [10] in the case of a carry-save implementation. At each step, our algorithm computes:

$$\begin{aligned} P[i] &= x_i Y + 2\left\langle \rho 2^{n-\alpha} \right\rangle_M \\ &\quad + 2\left(\left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}} + P^{(c)}[i+1]\right). \end{aligned}$$

According to this equation, $P[i]$ is always even when $x_i Y = 0$. Thus, by performing an additional step with $x_{-1} = 0$, we obtain an *even* number $P[-1]$ congruent to $2XY$ modulo M. Note that $P[-1]/2$ is smaller than or equal to $P[0]$ and easy to compute (a right shift of one position involves only wiring). Furthermore, performing the final modulo M correction with $P[-1]/2$ requires less hardware resources. Let us define:

$$\psi_{\max} = \begin{cases} \max\limits_{0 \le j < 2^4} \left\langle j \cdot 2^{n-2} \right\rangle_M & \text{if } \alpha = 2 \text{ and } n_{k-1} \ge 5, \\ \max\limits_{0 \le j < 2^3} \left\langle j \cdot 2^{n-1} \right\rangle_M & \text{if } \alpha = 1 \text{ and } n_{k-1} \ge 6. \end{cases}$$

$P[-1]/2$ is a high-radix carry-save number equal to $\langle XY \rangle_M$ or $\langle XY \rangle_M + M$, and the final modulo M reduction requires at most one subtraction in the following cases[1]:

- $\alpha = 2$ and $n_{k-1} \ge 5$;
- $\alpha = 1$, $n_{k-1} \ge 6$, and

$$\frac{2^{n-1} - 1 + 2^{n_0} + \ldots + 2^{n_0 + \ldots + n_{k-2}} + \psi_{\max}}{2} < M.$$

A proof of correctness of this modular multiplication scheme, summarized by Algorithm 1, is provided in Appendix. At each iteration, a new intermediate result $P[i]$ is computed in two steps. Let $\tilde{P}[i+1]$ be a high-radix carry-save number such that $\tilde{P}^{(s)}[i+1] = \left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}}$ and $\tilde{P}^{(c)}[i+1] = P^{(c)}[i+1]$. We first carry out the sum of the partial product $x_i Y$ and $2\tilde{P}[i+1]$ by means of small CRAs:

$$T[i] = 2\tilde{P}[i+1] + x_i Y.$$

By shifting the high-radix carry-save number $P[i+1]$, we define a new internal representation for $T[i]$ (Section II). The second step consists in adding $2\left\langle \rho \cdot 2^{n-\alpha} \right\rangle_M$ to $T[i]$, and in converting the result to the format of $P[i+1]$.

Algorithm 1 High-radix carry-save modulo M multiplication

Input: An n-bit modulus M such that $2^{n-1} < M < 2^n$, an r-bit number $X \in \mathbb{N}$, $Y \in \{0, \ldots, M-1\}$, and a parameter $\alpha \in \{1, 2\}$. $P[i]$ and $T[i]$ are high-radix carry-save numbers.
Output: $P = \langle XY \rangle_M$.

```
P[r] ← 0;
x_{-1} ← 0;
for i in r − 1 downto −1 do
    ρ ← P^(s)[i+1] div 2^{n−α};
    T[i] ← 2(⟨P^(s)[i+1]⟩_{2^{n−α}} + P^(c)[i+1]) + x_i Y;
    P[i] ← T[i] + 2⟨ρ · 2^{n−α}⟩_M;
end for
P ← P[−1]/2;
if P > M then
    P ← P − M;
end if
```

The main difficulty of the implementation arises from the left shift of the carry bits $P^{(c)}[i+1]$. Since $T[i]$ has a different encoding, it is necessary to perform a conversion. We suggest to compute a high-radix carry-save number $U[i]$ which has the same encoding as $P[i+1]$, and is equal to the sum of the carry bits of $T[i]$ and the output of the table (*i.e.* $2\left\langle \rho \cdot 2^{n-\alpha} \right\rangle_M$). Therefore, we perform the following operations at each iteration of Algorithm 1:

$$\begin{aligned} T[i] &\leftarrow 2\tilde{P}[i+1] + x_i Y, \\ U[i] &\leftarrow 2\left\langle \rho \cdot 2^{n-\alpha} \right\rangle_M + T^{(c)}[i], \\ P[i] &\leftarrow U[i] + T^{(s)}[i]. \end{aligned} \tag{6}$$

Example 3 *Let $n = 16$, $k = 4$, and $n_0 = n_1 = n_2 = n_3 = 4$. The high-radix carry-save number $T[i]$ contains three carry bits of respective weights 2^5, 2^9, and 2^{13} (recall the constraint introduced in Section II: the carry bit of the most significant digit*

[1]Note that the algorithm described in our preliminary work [12] does not satisfy this property and the final modular reduction depends on the high-radix number system and the modulus M.

is always equal to zero). We split $2\left\langle \rho \cdot 2^{n-\alpha} \right\rangle_M$ *into four* 4*-bit words and perform three additions to compute* $U[i]$ *(Figure 4).*

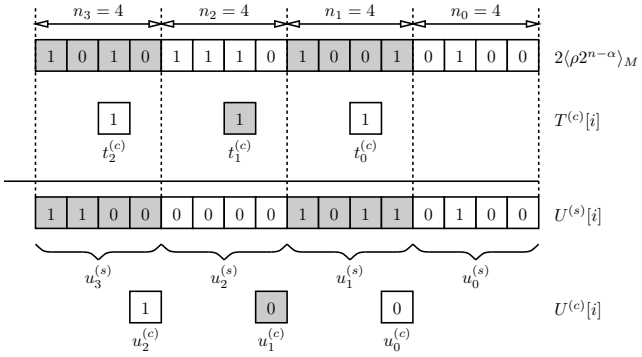


Fig. 4. Conversion of $T[i]$ by merging its carry bits with $2\left\langle \rho \cdot 2^{n-\alpha} \right\rangle_M$.

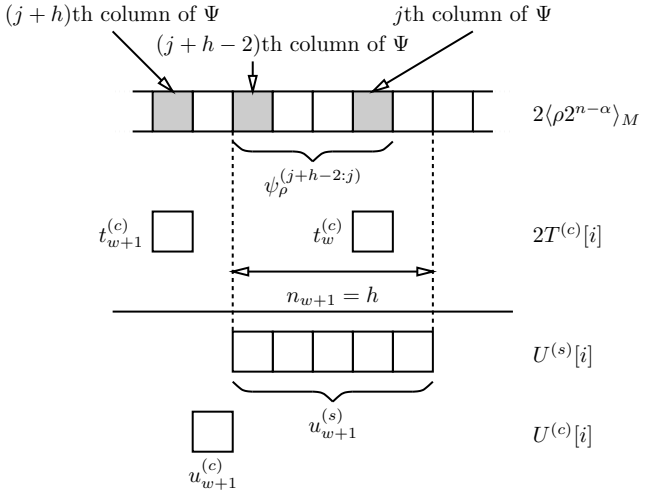


Fig. 5. Cost of the addition of a carry bit (1). In this example, h is equal to five.

IV. Choice of a High-Radix Carry-Save Number System

Let us represent the table involved in the modulo M correction by a matrix Ψ. Each line ψ_ρ of Ψ stores an n-bit number $\left\langle \rho \cdot 2^{n-\alpha} \right\rangle_M$. In the following, we will have to consider a subset of consecutive columns of Ψ. Let $\Psi^{(j+h:j)}$ be the matrix constituted of columns j to $j+h$ of Ψ. Each line of $\Psi^{(j+h:j)}$ contains an $(h+1)$-bit number $\psi_\rho^{(j+h:j)}$.

Example 4 *Let us consider the* 16*-bit modulus* $M = 54107$ *and assume that* $\alpha = 1$*. We have*

$$\Psi = \begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&1&1&0&0&1&0&1&0&0&1&0&1 \\ 1&0&1&0&1&1&0&0&1&0&1&0&0&1&0&1 \\ 0&1&0&1&1&0&0&1&0&1&0&0&1&0&1&0 \\ 0&0&0&0&0&1&0&1&1&1&1&0&1&1&1&1 \\ 1&0&0&0&0&1&0&1&1&1&1&0&1&1&1&1 \\ 0&0&1&1&0&0&1&0&1&0&0&1&0&1&0&0 \end{bmatrix}.$$

According to our notation, we have for instance:

$$\Psi^{(6:3)} = \left(\psi_\rho^{(6:3)}\right) = \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 1&0&0&1 \\ 1&0&0&1 \\ 0&0&1&0 \\ 1&0&1&1 \\ 1&0&1&1 \\ 0&1&0&1 \end{bmatrix}. \quad (7)$$

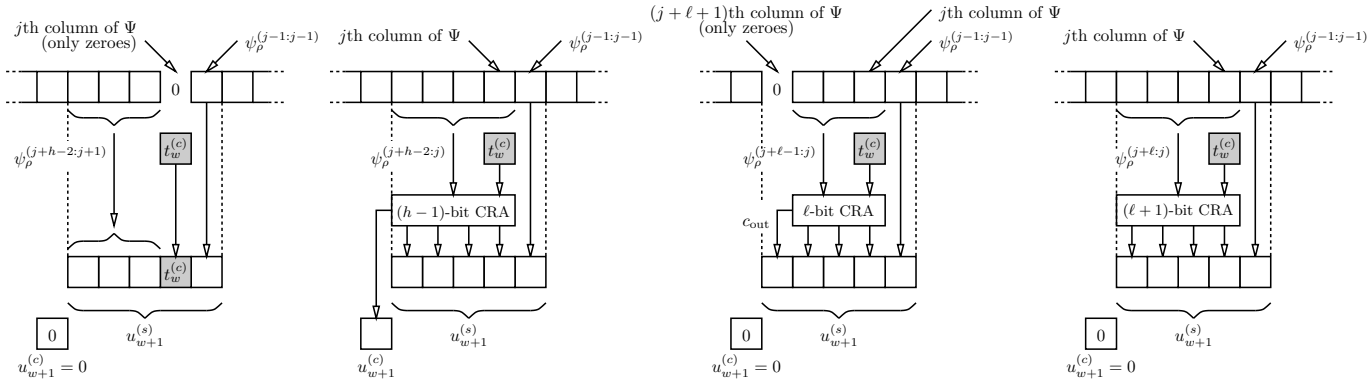
It is worth noticing that the amount of hardware required to compute $U[i]$ depends on the modulus M and the encoding of $P[i]$. For instance, if a column of Ψ contains only zeroes, it can be replaced by a carry bit at no extra cost. We propose an algorithm which selects a high-radix carry-save number system minimizing the hardware overhead introduced by the computation of $U[i]$ (Equation (6)). Assume that we want to merge $t_w^{(c)}$ with the jth column and $t_{w+1}^{(c)}$ with the $(j+h)$th column of Ψ, and recall that the carry bits of $T[i]$ are left-shifted compared to those of $P[i]$ and $U[i]$ (Figure 5). Therefore, we compute a digit u_{w+1} such that

$$u_{w+1}^{(c)} 2^h + u_{w+1}^{(s)} = 2\Big(\underbrace{\psi_\rho^{(j+h-2:j)}}_{h-1 \text{ bits}} + t_w^{(c)}\Big) + \psi_\rho^{(j-1:j-1)}. \quad (8)$$

This operation involves at most a $(h-1)$-bit CRA. However it is unlikely that there are very long strings of consecutive ones in matrix ψ (see below). Let us denote by $\#\left(\psi_\rho^{(j+h-2:j)}\right)$ the length of the longest string of consecutive 1's starting from the least significant bit of $\psi_\rho^{(j+h-2:j)}$. Then, the following cases may occur according to $\ell = \max_\rho \#\left(\psi_\rho^{(j+h-2:j)}\right)$:

- If $\ell = 0$, the jth column of Ψ contains only zeroes and can be replaced by $t_w^{(c)}$ at no extra cost. More formally, we have (Figure 6a):
$$u_{w+1}^{(s)} = 4\psi_\rho^{(j+h-2:j+1)} + 2t_w^{(c)} + \psi_\rho^{(j-1:j-1)},$$
$$u_{w+1}^{(c)} = 0.$$
- If $\ell = h-1$, the addition requires a $(h-1)$-bit CRA which generates an output carry bit $u_{w+1}^{(c)}$ (see Equation (8) and Figure 6b). Since this bit will be added to a few bits of $T^{(s)}[i]$ in order to compute $p_{w+2}^{(s)}[i]$, we raise a flag which indicates this carry propagation.
- When $0 < \ell < h-1$, an ℓ-bit CRA computes the sum and generates an output carry c_{out}. If the $(j+\ell)$th column of Ψ stores only zeroes, we replace it by c_{out} (Figure 6c). Otherwise, we need an OR gate to add c_{out} to the $(j+\ell)$th column. Since we target FPGA applications, a more efficient solution consists in taking advantage of the dedicated carry logic to perform this operation and we add $t_w^{(c)}$ to $\psi_\rho^{(j+\ell:j)}$ by means of an $(\ell+1)$-bit CRA (Figure 6d). Note that $u_{w+1}^{(c)}$ is *always* equal to zero.

Let us try to estimate what values of ℓ we can expect in practice. If we consider that the bits in matrix ψ can be viewed as "random" bits, with equal probability for 0 as for 1, then the average value of ℓ will be $N(h)/2^{h-1}$, where $N(h)$ is the sum of the lengths of the strings of ones that start from the rightmost bit in all possible $(h-1)$-bit strings. Since we obviously have $N(1) = 1$ and $N(h) =$



(a) Only zeroes in the jth column of Ψ (b) $\ell = h - 1$ (c) $\ell < h - 1$ and the $(j + \ell + 1)$th column of Ψ contains only zeroes (d) $\ell < h - 1$

Fig. 6. Cost of the addition of a carry bit (2).

$2N(h-1)+1$, we immediately deduce that the average value of ℓ is $1 - 1/2^{h-1}$ (*i.e.* less than 1).

Example 5 (Example 4 continued) *Assume that we want to add carry bits to the 3rd and 8th columns of Ψ (i.e. $j = 3$ and $h = 5$). We have to consider the matrix $\Psi^{(6:3)}$ given by Equation (7) and easily determine that $\ell = 2$. Thus, we have to examine the third column of $\Psi^{(6:3)}$ in order to compute the width of the CRA. Since this column contains a non-null element, we need an $(\ell+1)$-bit CRA (see Figure 6d).*

Let us now build a directed acyclic graph as follows:

- Each node represents a column of the matrix Ψ.
- The weight of the edge between nodes j and $j+h$ is given by the width of the CRA responsible for the addition of $\psi_\rho^{(j+h-2:j)}$ and $t_w^{(c)}$ (Equation (8)), as well as the flag which indicates a carry propagation.

A shortest path of this graph defines the high-radix carry-save representation minimizing the hardware overhead introduced by the computation of $U[i]$ for a given modulus M. The algorithm requires two parameters to control the size of the CRAs:

- Since we want to perform a modular multiplication by means of small CRAs, we have to provide the algorithm with a constraint on the maximal number of positions $w_{\max}$ between two consecutive carry bits (without this constraint, we would for instance have an edge from the first node to the last node).
- The minimal distance between to consecutive carries $w_{\min}$ should be greater than or equal to two. It guarantees that the smallest building block is a 2-bit CRA.

Algorithm 2 describes a way to build this graph. After having computed Ψ, we have to determine to which columns the most significant bit of $T[i]$ can be added. We denote by $j_{\max}$ the upper index. Recall that, when $\alpha = 2$, we assume that the most significant sum word of $P[i]$ contains at least $n_{k-1} = 5$ bits. Thus, we deduce that $j_{\max} = n - 2$ (Figure 7a). The most significant sum word of $U[i]$ is computed as follows:

$$u_{k-1}^{(s)} = 2\left(\psi_\rho^{(n:j_{\max})} + t_{k-2}^{(c)}\right) + \psi_\rho^{(j_{\max}-1:j_{\max}-1)}.$$

When $\alpha = 1$, we have $n_{k-1} \geq 6$ and $j_{\max} = n-3$ (Figure 7b). It is sometimes possible to relax the constraint on n_{k-1}: it suffices that the addition of $t_{k-2}^{(c)}$ to $\psi_\rho^{(n:j_{\max})}$ does not generate an output carry (see the proof in Appendix for details). This condition is satisfied if the length of the longest string of consecutive 1's starting from the $j_{\max}$ column of Ψ is smaller than or equal to $n - j_{\max}$. We have to distinguish three cases to build the graph:

- The first carry bit $t_0^{(c)}$ can be added to any column whose index belongs to $\{2, \ldots, w_{\max}+1\}$. We create an edge of weight zero between the first node and the nodes associated with such columns.
- The most significant carry bit $t_{k-2}^{(c)}$ belongs to the set $\{n - w_{\max}+1, \ldots, j_{\max}$. Let $h \in \{w_{\min}, \ldots, w_{\max}-1\}$. There is therefore an edge between nodes j and n if $j + h \geq n$.
- There is an edge between nodes j and $j+h$, where $h \in \{w_{\min}, \ldots, w_{\max}\}$, if $j + h \leq j_{\max}$.

The next step consists in finding a shortest path in the graph. In order to minimize the critical path, we suggest to remove all edges whose carry propagation flag is set to one. If there is no path between nodes 1 and n in this pruned graph, we have to consider the full graph.

Example 6 (Example 4 continued) *Let us apply Algorithm 2 to our 16-bit example. First, we note that adding a carry bit to the 15th column of the matrix does not generate an output carry and we set $j_{max} = 15$. Then, we build the graph illustrated on Figure 8 according to Algorithm 2. The c flag on the edge between nodes j and $j+h$ indicates that adding a carry bit to the jth column of Ψ generates an output carry bit $u_j^{(c)}$. After having removed all edges labelled with a c flag and nodes without predecessor or successor, we obtain a pruned graph (Figure 9). Thus, $P[i]$ is a 4-digit word with $n_0 = n_1 = n_2 = 4$ and $n_3 = 6$ (Figure 10). Since $n = 16$ and $\psi_{max} = (1010110010100101)_2 = 44197$, we have:*

$$\frac{2^{n-1} - 1 + 2^{n_0} + 2^{n_0+n_1} + 2^{n_0+n_1+n_2} + \psi_{max}}{2} = \frac{2^{15} + 2^4 + 2^8 + 2^{12} + 44197}{2} = 40666 < M,$$

and $\alpha = 1$ is a valid choice. Note that, if the above equality is not satisfied, we have to build a new graph with $\alpha = 2$. Recall that there is always a solution for $\alpha = 2$ and $n_{k-1} \geq 5$.

Once the high-radix carry-save representation is known, the automatic generation of a VHDL description of the modulo M multiplier is rather straightforward. The computation of $T[i]$

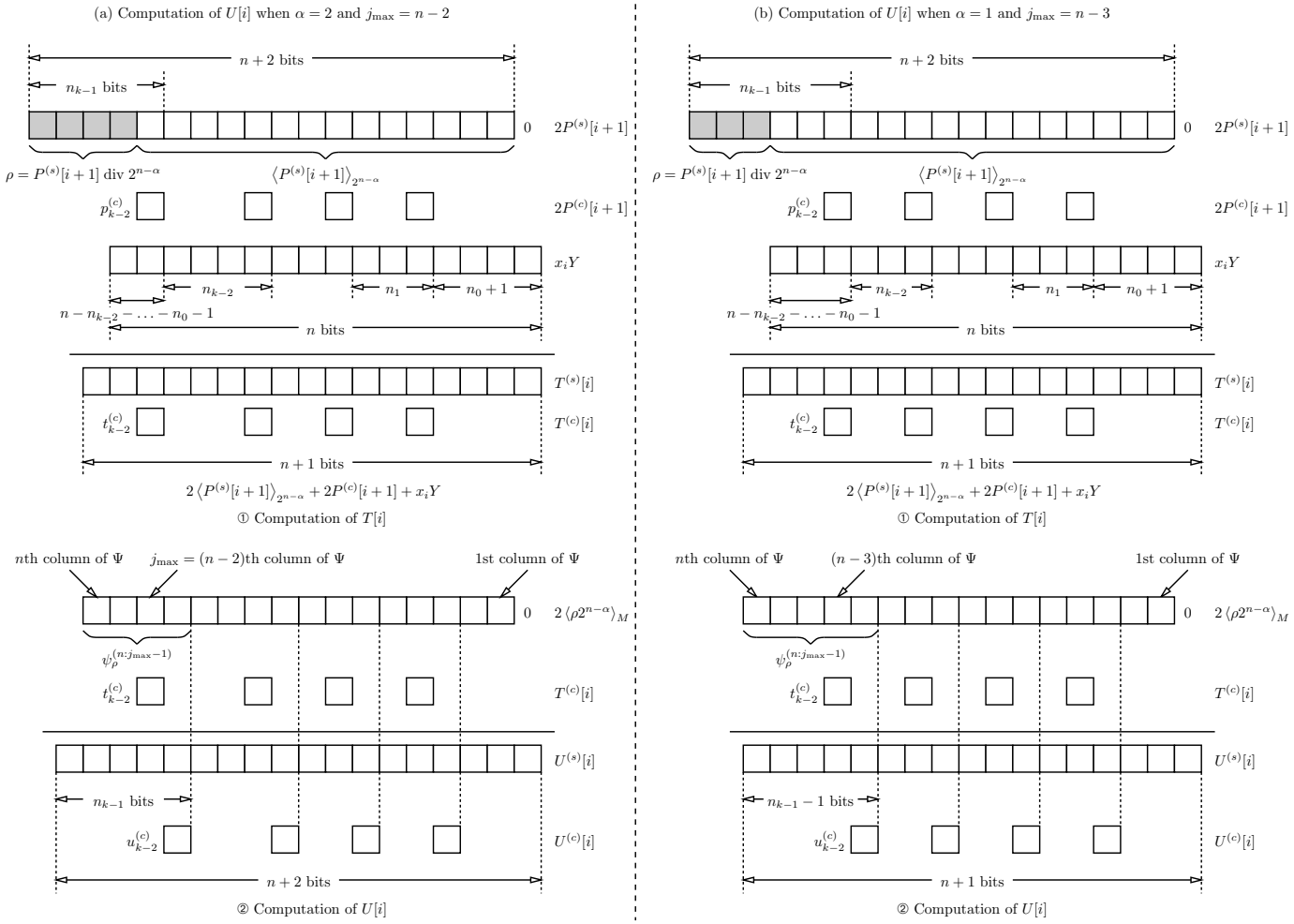


Fig. 7. Cost of the addition of a carry bit (3).

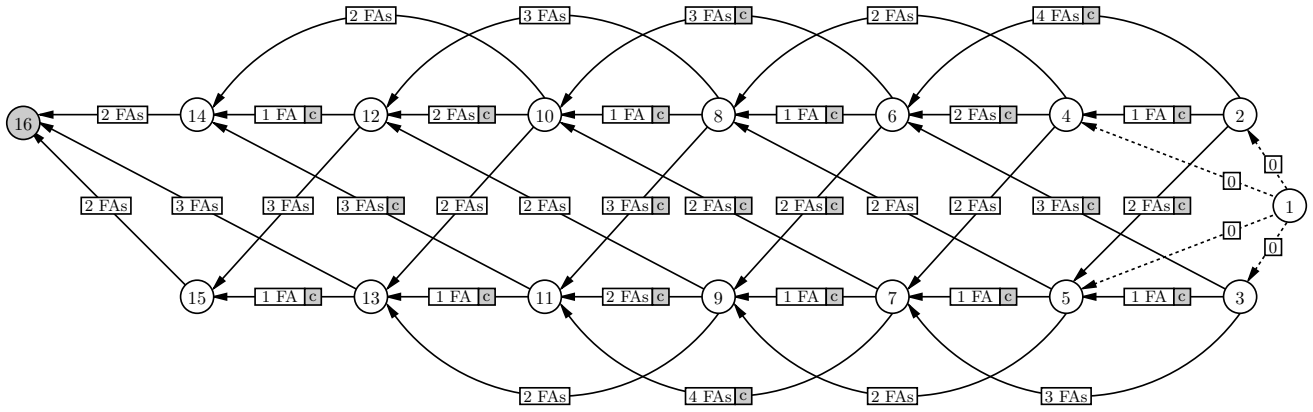


Fig. 8. Choice of a high-radix carry-save representation for the 16-bit modulus $M = 54107$ (1).

involves k CRAs of respective widths $n_0 + 1$, n_1, …, $n_k - 2$, and $n - n_{k-2} - \ldots - n_0 - 1$ (Figure 7). Each edge of the graph encodes the size of the CRA determining a digit of $U[i]$ and the carry propagation flag indicates whether a carry bit $u_j^{(c)}$ is necessary or not. Finally, k CRAs of widths n_0, …, $n_k - 1$ allow one to add $T^{(s)}[i]$ to $U[i]$.

V. Implementation Results

In order to compare our algorithm against modular multipliers published in the open literature, we wrote a VHDL code generator whose inputs are a modulus M and $w_{\max}$ (maximal number of positions between two consecutive carry bits, see Section IV). Our tool returns a structural VHDL description of a high-radix carry-save multiplier, as well as scripts to automatically place-and-route the design and collect area and timing informations. This tool also

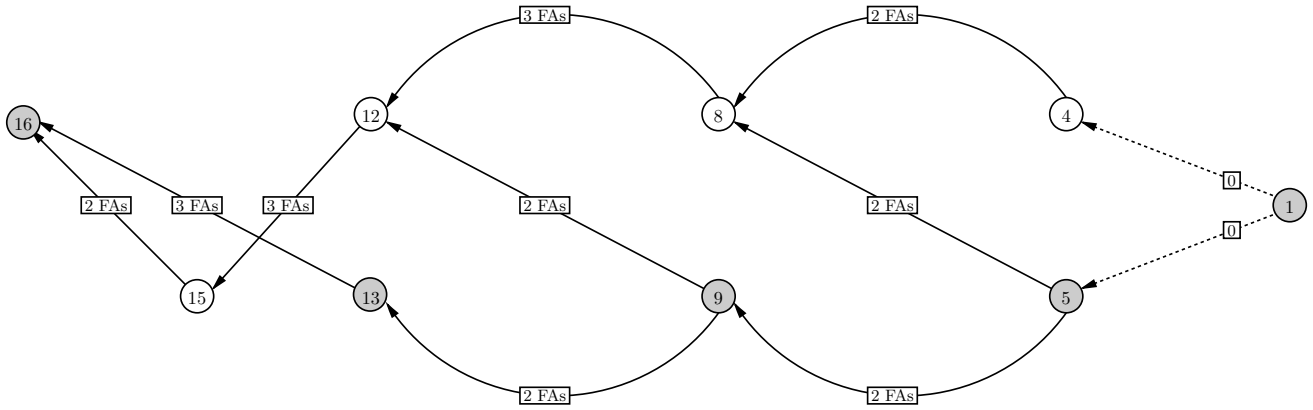


Fig. 9. Choice of a high-radix carry-save representation for the 16-bit modulus $M = 54107$ (2). Shaded nodes belong to the shortest path.

Algorithm 2 Selection of a high-radix carry-save number system.

Input: An n-bit modulus M, $w_{\max}$, and $w_{\min}$ such that $w_{\max} \geq w_{\min} \geq 2$. A parameter $\alpha \in \{1, 2\}$.

Output:

1: Compute the matrix Ψ according to α;
2: **if** $\alpha = 2$ **then**
3: $j_{\max} \leftarrow n - 2$;
4: **else**
5: $j_{\max} \leftarrow n - 3$;
6: $j \leftarrow n - 2$
7: **while** $\#\left(\psi_\rho^{(n:j)}\right) \leq n - j$ **do**
8: $j_{\max} \leftarrow j$;
9: $j \leftarrow j + 1$;
10: **end while**
11: **end if**
12: **for** $j = 2$ to $w_{\max}$ **do**
13: Create an edge of weight 0 between nodes 1 and j;
14: **end for**
15: **for** $j = 2$ to $j_{\max}$ **do**
16: **for** $h = w_{\min}$ to $w_{\max}$ **do**
17: **if** $j + h \geq n$ and $h < w_{\max}$ **then**
18: $\ell \leftarrow \max_\rho \#\left(\psi_\rho^{(n:j)}\right)$;
19: Create an edge between nodes j and n;
20: Compute the weight of the edge (see Figure 6);
21: $h \leftarrow w_{\max} + 1$ (exit the loop);
22: **else if** $j + h \leq j_{\max}$ **then**
23: $\ell \leftarrow \max_\rho \#\left(\psi_\rho^{(j+h-2:j)}\right)$;
24: Create an edge between nodes j and $j + h$;
25: Compute the weight of the edge (see Figure 6);
26: **end if**
27: **end for**
28: **end for**

generates a VHDL description of two architectures proposed by other researchers. The first one, described by Peeters *et al.* in [10], is summarized by Algorithm 3. Intermediate results are carry-save numbers denoted by $(C[i], S[i])$. At each step, a CRA computes the sum k of the three most significant bits of $C[i + 1]$ and $S[i+1]$. This four-bit word addresses a table storing $\left\langle k \cdot 2^{n-2} \right\rangle_M$, $0 \leq k \leq 15$. Thanks to an additional iteration with $x_{-1} = 0$, this algorithm returns a carry-save number $(C[-1], S[-1])$ which is

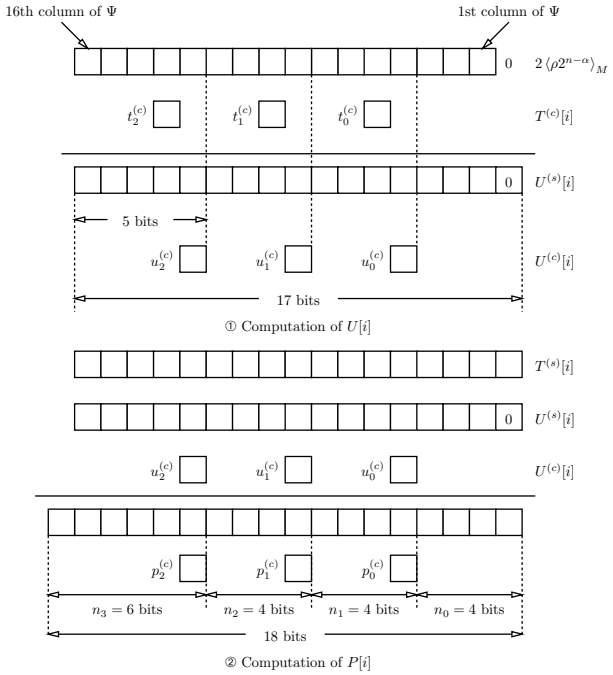


Fig. 10. Choice of a high-radix carry-save representation for the 16-bit modulus $M = 54107$ (3).

smaller than $2M$. Since our multiplier satisfies the same property, conversion in a non-redundant number system is performed with the same operator[2]. We will therefore only consider iteration stages in our experiments in order to compare high-radix carry-save and carry-save number systems.

Amanor *et al.* introduced a carry-save architecture optimized for modular multiplication on FPGAs in [13]. The authors assume that both M and Y are known at design time. This hypothesis allows for the design of an iteration stage embedding a single CSA and a table addressed by the most significant bit of x_i, $C[i + 1]$, and $S[i + 1]$ (Algorithm 4). They show that the sum of the most significant bits of $C[i+1]$ and $S[i+1]$ is always a two-bit number. Therefore the table only contains eight values. Unfortunately, the authors did not address the final conversion issue. However, since

[2]Our approach further reduces the wiring since high-radix carry-save numbers involve less carry bits than carry-save numbers.

Algorithm 3 Peeters *et al.*'s modulo M multiplication [10].

Input: An n-bit modulus M such that $2^{n-1} < M < 2^n$, an r-bit number $X \in \mathbb{N}$, and $Y \in \{0, \ldots, M-1\}$. We assume that $x_{-1} = 0$.
Output: $P = \langle XY \rangle_M$.
$C[r] \leftarrow 0$; $S[r] \leftarrow 0$;
for i in $r-1$ downto -1 **do**
  $k \leftarrow C[i+1] \text{ div } 2^{n-2} + S[i+1] \text{ div } 2^{n-2}$;
  $T[i] \leftarrow x_i Y + 2\left\langle k \cdot 2^{n-2} \right\rangle_M$;
  $(C[i], S[i]) \leftarrow 2(\langle C[i+1] \rangle_{2^{n-2}} + \langle S[i+1] \rangle_{2^{n-2}}) + T[i]$;
end for
$P \leftarrow (S[-1] + C[-1])/2$;
if $P > M$ **then**
  $P \leftarrow P - M$;
end if

$C[i+1] \text{ div } 2^{n-1} + S[i+1] \text{ div } 2^{n-1} \leq 3$, we deduce that:

$$\begin{aligned} C[i] + S[i] &= (C[i+1] \text{ div } 2^{n-1} + S[i+1] \text{ div } 2^{n-1}) \cdot 2^{n-1} \\ &\quad + \langle C[i+1] \rangle_{2^{n-1}} + \langle S[i+1] \rangle_{2^{n-1}} \\ &\leq 3 \cdot 2^{n-1} + 2 \cdot (2^{n-1} - 1) = 2^{n+1} + 2^{n-1} - 2. \end{aligned}$$

Since M belongs to $\{2^{n-1}+1, \ldots, 2^n - 1\}$, $C[i] + S[i]$ may be greater than $2M$ and Algorithm 4 requires more hardware resources than our algorithm or Peeters *et al.*'s scheme to perform a conversion.

Algorithm 4 Amanor *et al.*'s modulo M multiplication [13].

Input: An n-bit modulus M such that $2^{n-1} < M < 2^n$, an r-bit number $X \in \mathbb{N}$, and $Y \in \mathbb{N}$.
Output: $P = \langle XY \rangle_M$.
$C[r] \leftarrow 0$; $S[r] \leftarrow 0$;
for i in $r-1$ downto 0 **do**
  $k \leftarrow 2(C[i+1] \text{ div } 2^{n-1} + S[i+1] \text{ div } 2^{n-1})$;
  $T[i] \leftarrow \left\langle k \cdot 2^{n-1} + x_i Y \right\rangle_M$;
  $(C[i], S[i]) \leftarrow 2(\langle C[i+1] \rangle_{2^{n-1}} + \langle S[i+1] \rangle_{2^{n-1}}) + T[i]$;
end for
$P \leftarrow \langle C[0] + S[0] \rangle_M$;

Figure 11 describes place-and-route results on a Xilinx Spartan-3 FPGA. In these experiments, the moduli are 256-bit randomly generated primes. Compared against Algorithm 3, we observe that:

- Our high-radix carry-save architecture allows us to significantly reduce the number of slices, while only slightly augmenting the critical path. At the price of a longer critical path, we are able to further diminish the area by increasing the parameter w_{max}. Note that conversion from (high-radix) carry-save to unsigned binary integer is usually based on pipelined CRAs (see for instance [3]). Depending on the trade-off between area and delay, this operator can be slower than an iteration stage based on (high-radix) carry-save arithmetic.
- The area of our operator is less sensitive to the choice of M. This is mainly related to the architecture of Xilinx FPGAs: in most cases, $\alpha = 1$ and each operator embeds a table addressed by three bits. Since our target FPGA embeds four-input LUTs, this table is embedded within the slices of the adder computing $P[i]$ [11]. Since four bits address the table of Algorithm 3, additional LUTs are requested. Their amount depends on the modulus M: if ψ_M contains null or identical columns, synthesis tools are able to simplify the design.

For the moduli considered in these experiments, high-radix carry-save multipliers have roughly the same area as the operator proposed by Amanor *et al.* in [13]. Recall that a CSA requires twice the number of slices of a CRA on our target FPGA family. Since moduli involved in these experiments require only three bits to perform a modulo M reduction, our architecture is mainly based on two CRAs. High-radix carry-save representations enable here the design of a more versatile modular multiplier (both X and Y are inputs) with the same number of slices.

Table III summarizes further results obtained with a Spartan-3 FPGA. We generated 100 prime moduli for each experiment, and reported the interval in which lie the area and delay ratios between our proposal and Algorithms 3 and 4. These experiments indicate that our approach always allow one to select a prime number for which reduces the circuit area without increasing the critical path.

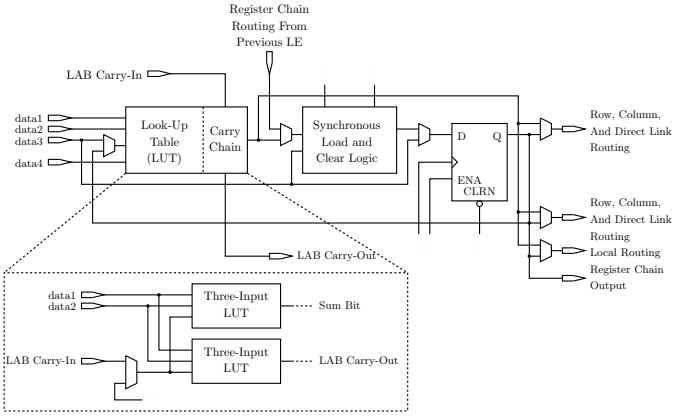


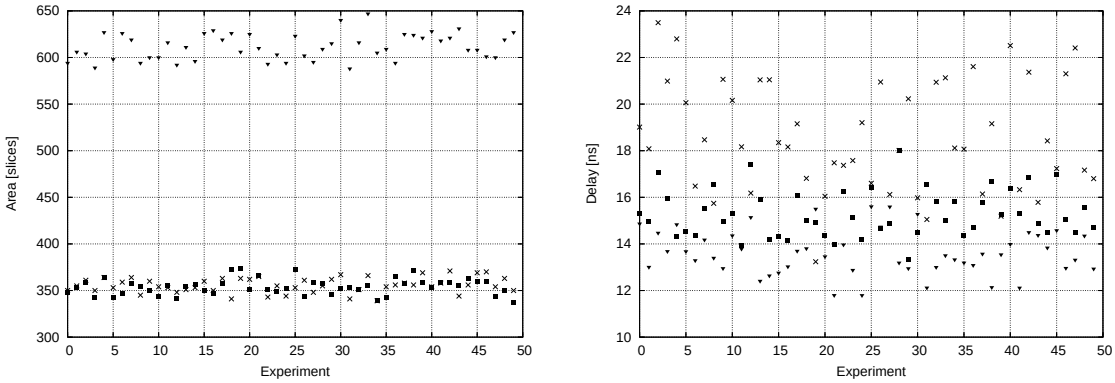
Fig. 12. Simplified diagram of a LE in arithmetic mode (Cyclone II family).

Table IV digests experiment results involving an Altera Cyclone II FPGA. Figure 12 describes a Logic Element (LE), which is the smallest unit of configurable logic in the Cyclone II architecture. Each LE includes a four-input LUT, a storage element, as well as dedicated carry logic, and operates in normal mode or arithmetic mode. CRAs are based on LEs in arithmetic mode, in which the LUT implements two three-input function generators. It is therefore impossible to store ψ_M within LUTs of a CRA. This explains why our algorithm leads to slightly smaller area savings for this FPGA family. On Cyclone-II devices, CSA operators are significantly faster; however, conversion to a non-redundant number system involves pipelined CRAs. If this operator is based on 32-bit blocks, our high-radix carry-save iteration stage has a slower critical path. In this case, our approach leads to smaller modular multipliers than CSA schemes, without impacting on computation time.

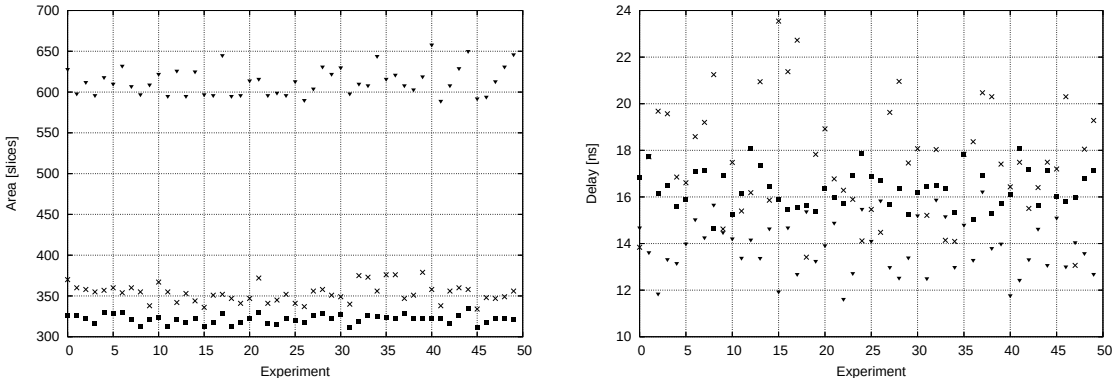
VI. Conclusion

We proposed an algorithm to automatically generate VHDL descriptions of modular multipliers for FPGAs. The main originality of our approach is the selection of an optimal high-radix carry-save encoding of intermediate results according to a given modulus M. High-radix carry-save number systems take

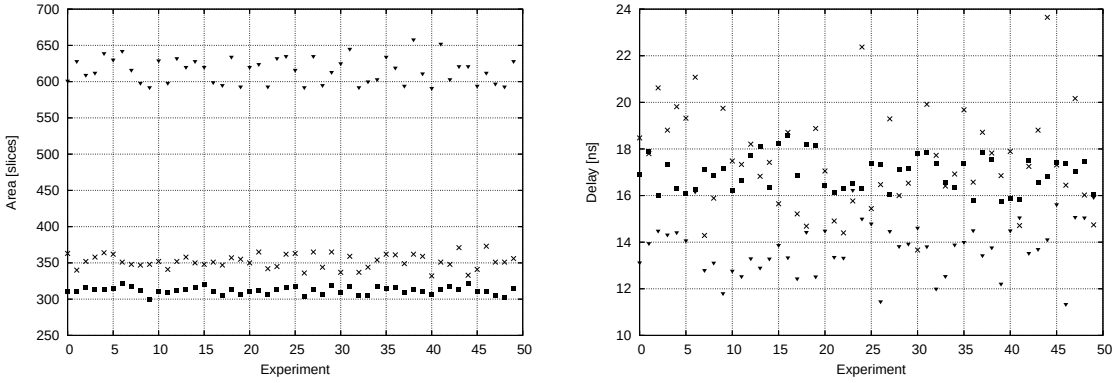
■ Proposed algorithm; ▼ Algorithm 3; × Algorithm 4.



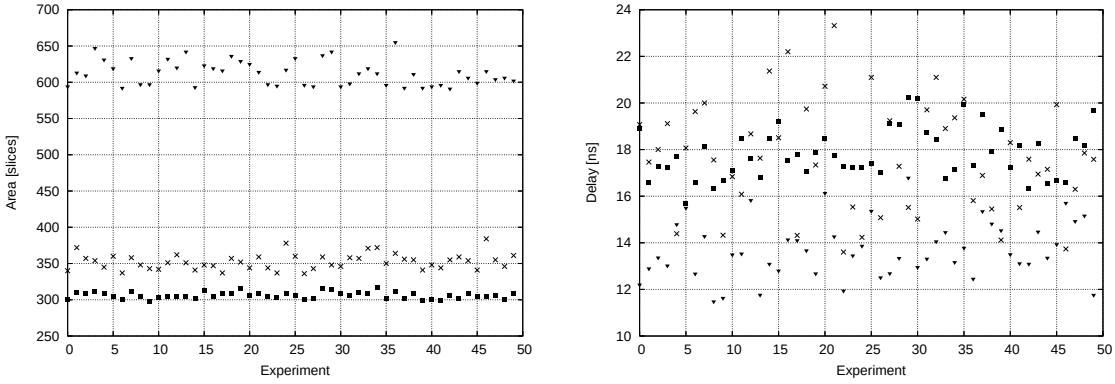
(a) Area and delay comparisons for $n = 256$ and $w_{\max} = 8$



(b) Area and delay comparisons for $n = 256$ and $w_{\max} = 16$



(c) Area and delay comparisons for $n = 256$ and $w_{\max} = 24$



(d) Area and delay comparisons for $n = 256$ and $w_{\max} = 32$

Fig. 11. Area and delay of modular multipliers on a Spartan-3 FPGA. 50 prime moduli were randomly generated for each experiment.

TABLE III

AREA AND DELAY RATIOS BETWEEN OUR PROPOSAL AND ALGORITHMS 3 AND 4 ON A SPARTAN-3 FPGA. 100 PRIME MODULI WERE RANDOMLY GENERATED FOR EACH EXPERIMENT.

N	w_{max}	Peeters *et al.* [10]		Amanor *et al.* [13]	
		Area of Algorithm 1 / Area of Algorithm 3	Delay of Algorithm 1 / Delay of Algorithm 3	Area of Algorithm 1 / Area of Algorithm 4	Delay of Algorithm 1 / Delay of Algorithm 4
64	**8**	$[0.45, 0.68]$	$[0.89, 1.45]$	$[0.77, 1.12]$	$[1.10, 1.62]$
	16	$[0.39, 0.58]$	$[0.97, 1.49]$	$[0.73, 0.97]$	$[1.12, 1.84]$
128	**8**	$[0.48, 0.68]$	$[0.99, 1.32]$	$[0.89, 1.28]$	$[0.99, 1.38]$
	16	$[0.44, 0.55]$	$[0.99, 1.32]$	$[0.79, 0.96]$	$[0.99, 1.44]$
	32	$[0.43, 0.53]$	$[1.00, 1.44]$	$[0.77, 0.91]$	$[1.01, 1.61]$
	48	$[0.40, 0.50]$	$[1.04, 1.69]$	$[0.74, 0.89]$	$[1.11, 1.79]$
160	**8**	$[0.51, 0.64]$	$[0.98, 1.15]$	$[0.90, 1.09]$	$[0.99, 1.23]$
	16	$[0.48, 0.56]$	$[0.99, 1.19]$	$[0.84, 0.96]$	$[0.99, 1.32]$
	32	$[0.44, 0.53]$	$[1.04, 1.36]$	$[0.78, 0.94]$	$[1.04, 1.45]$
	48	$[0.36, 0.52]$	$[1.11, 1.56]$	$[0.79, 0.89]$	$[1.18, 1.63]$
192	**8**	$[0.53, 0.63]$	$[0.57, 1.38]$	$[0.92, 1.12]$	$[0.67, 1.49]$
	16	$[0.48, 0.55]$	$[0.62, 1.47]$	$[0.82, 0.97]$	$[0.82, 1.59]$
	24	$[0.46, 0.53]$	$[0.55, 1.49]$	$[0.83, 0.98]$	$[0.85, 1.69]$
	32	$[0.46, 0.52]$	$[0.65, 1.46]$	$[0.79, 0.93]$	$[0.94, 1.66]$
256	**8**	$[0.54, 0.63]$	$[0.89, 1.48]$	$[0.93, 1.11]$	$[0.59, 1.27]$
	16	$[0.49, 0.55]$	$[0.94, 1.45]$	$[0.85, 0.98]$	$[0.67, 1.34]$
	24	$[0.48, 0.53]$	$[0.99, 1.59]$	$[0.83, 0.97]$	$[0.71, 1.38]$
	32	$[0.47, 0.53]$	$[1.01, 1.68]$	$[0.79, 0.94]$	$[0.66, 1.35]$

TABLE IV

AREA AND DELAY RATIOS BETWEEN OUR PROPOSAL AND ALGORITHMS 3 AND 4 ON A CYCLONE II FPGA. 100 PRIME MODULI WERE RANDOMLY GENERATED FOR EACH EXPERIMENT.

N	w_{max}	Peeters *et al.* [10]		Amanor *et al.* [13]	
		Area of Algorithm 1 / Area of Algorithm 3	Delay of Algorithm 1 / Delay of Algorithm 3	Area of Algorithm 1 / Area of Algorithm 4	Delay of Algorithm 1 / Delay of Algorithm 4
64	**8**	$[0.63, 0.77]$	$[1.45, 1.87]$	$[1.14, 1.36]$	$[1.92, 2.54]$
	16	$[0.60, 0.66]$	$[1.58, 2.02]$	$[1.06, 1.19]$	$[2.18, 2.75]$
128	**8**	$[0.67, 0.74]$	$[1.42, 1.87]$	$[1.14, 1.30]$	$[1.77, 2.49]$
	16	$[0.62, 0.65]$	$[1.63, 2.04]$	$[1.07, 1.15]$	$[2.09, 2.85]$
	32	$[0.58, 0.63]$	$[1.85, 2.73]$	$[1.01, 1.09]$	$[2.18, 3.79]$

advantage of dedicated carry logic available in almost all FPGA families and reduce the amount of interconnects. Therefore, our approach allows us to significantly reduce the area of modular multipliers.

ACKNOWLEDGMENT

The authors are grateful to the anonymous referees for their valuable comments. The work described in this paper has been supported in part by the New Energy and Industrial Technology Development Organization (NEDO), Japan, and by the Swiss National Science Foundation through the *Advanced Researchers* program while Jean-Luc Beuchat was at *École Normale Supérieure de Lyon* (grant PA002–101386). The authors would like to thank F. de Dinechin et J. Detrey for administrating the CAD tool servers on which experiments described in this paper were performed.

REFERENCES

[1] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[2] G. R. Blakley, "A computer algorithm for calculating the product ab modulo m," *IEEE Trans. Comput.*, vol. C–32, no. 5, pp. 497–500, 1983.

[3] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.

[4] C. K. Koç and C. Y. Hung, "Carry-save adders for computing the product AB modulo N," *Electronics Letters*, vol. 26, no. 13, pp. 899–900, June 1990.

[5] ——, "A fast algorithm for modular reduction," *IEE Proceedings: Computers and Digital Techniques*, vol. 145, no. 4, pp. 265–271, July 1998.

[6] N. Takagi and S. Yajima, "Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem," *IEEE Trans. Comput.*, vol. 41, no. 7, pp. 887–891, July 1992.

[7] N. Takagi, "A radix-4 modular multiplication hardware algorithm for modular exponentiation," *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 949–956, Aug. 1992.

[8] Y.-J. Jeong and W. P. Burleson, "VLSI array algorithms and architectures for RSA modular multiplication," *IEEE Trans. VLSI Syst.*, vol. 5, no. 2, pp. 211–217, June 1997.

[9] S. Kim and G. E. Sobelman, "Digit-serial modular multiplication using skew-tolerant domino CMOS," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2. IEEE Computer Society, 2001, pp. 1173–1176.

[10] E. Peeters, M. Neve, and M. Ciet, "XTR implementation on reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems –*

CHES 2004, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., no. 3156. Springer, 2004, pp. 386–399.
[11] J.-L. Beuchat and J.-M. Muller, "Modulo m multiplication-addition: Algorithms and FPGA implementation," *Electronics Letters*, vol. 40, no. 11, pp. 654–655, May 2004.
[12] R. Beguenane, J.-L. Beuchat, J.-M. Muller, and S. Simard, "Modular multiplication of large integers on FPGA," in *Proceedings of the 39th Asilomar Conference on Signals, Systems & Computers*. IEEE Signal Processing Society, 2005.
[13] D. N. Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler, "Efficient hardware architectures for modular multiplication on FPGAs," in *Proceedings of FPL 2005*, 2005, pp. 539–542.

APPENDIX

This Appendix aims at proving the correctness of Algorithm 1. We proceed in three steps: after establishing a property of the modulo M correction considered in this paper, we show that $P^{(s)}[i]$ is an $(n+2)$-bit number. We conclude by computing a bound on $P[-1]$ which indicates that $P[-1]/2 < 2M$. This proof also provides the reader with all the technical details requested to implement the algorithm or an automatic code generator.

A. A Property of Modulo M Correction

The first step consists in establishing a property which will allow us to compute bounds on $P[i]$. Let $\gamma = 2^n - 2^{n-4} = 2^{n-1} + 2^{n-2} + 2^{n-3} + 2^{n-4}$ and $M \in \{2^{n-1}, \ldots, 2^n - 1\}$. Then,

$$\left\langle k \cdot 2^{n-2} \right\rangle_M < \gamma, \ \forall k \in 0, \ldots, 2^4 - 1. \tag{9}$$

The proof is straightforward if the modulus M is smaller than or equal to γ. Let us assume now that $M = \gamma + \beta$, where β satisfies the following inequality:

$$1 \leq \beta \leq 2^{n-4} - 1.$$

For $k \in \{0, 1, 2, 3\}$, we easily check that $\left\langle k \cdot 2^{n-2} \right\rangle_M = k \cdot 2^{n-2} < \gamma$. Since $\langle 2^n \rangle_M = 2^n - M$, we obtain:

$$\left\langle 4 \cdot 2^{n-2} \right\rangle_M = 2^{n-4} - \beta < \gamma.$$

Consequently, we have:

$$\begin{aligned}
\left\langle 5 \cdot 2^{n-2} \right\rangle_M &= 2^{n-2} + 2^{n-4} - \beta < \gamma, \\
\left\langle 6 \cdot 2^{n-2} \right\rangle_M &= 2^{n-1} + 2^{n-4} - \beta < \gamma, \text{ and} \\
\left\langle 7 \cdot 2^{n-2} \right\rangle_M &= 2^{n-1} + 2^{n-2} + 2^{n-4} - \beta < \gamma.
\end{aligned}$$

For $k = 8$, the following modulo M operation has to be carried out:

$$\left\langle 8 \cdot 2^{n-2} \right\rangle_M = \left\langle 2^n + 2^{n-4} - \beta \right\rangle_M.$$

Since $M < 2^n + 1 \leq 2^n + 2^{n-4} - \beta \leq 2^n + 2^{n-4} - 1 < 2M$, we deduce that:

$$\left\langle 8 \cdot 2^{n-2} \right\rangle_M = 2^n + 2^{n-4} - \beta - M = 2^{n-3} - 2\beta.$$

Thus, we have:

$$\begin{aligned}
\left\langle 9 \cdot 2^{n-2} \right\rangle_M &= 2^{n-2} + 2^{n-3} - 2\beta < \gamma, \\
\left\langle 10 \cdot 2^{n-2} \right\rangle_M &= 2^{n-1} + 2^{n-3} - 2\beta < \gamma, \text{ and} \\
\left\langle 11 \cdot 2^{n-2} \right\rangle_M &= 2^{n-1} + 2^{n-2} + 2^{n-3} - 2\beta < \gamma.
\end{aligned}$$

A modulo M reduction is again required for $k = 12$. Since

$$M < 2^n + 2 \leq 2^n + 2^{n-3} - 2\beta \leq 2^n + 2^{n-3} - 2 < 2M,$$

we obtain:

$$\begin{aligned}
\left\langle 12 \cdot 2^{n-2} \right\rangle_M &= \left\langle 2^n + 2^{n-3} - 2\beta \right\rangle_M \\
&= 2^n + 2^{n-3} - 2\beta - M \\
&= 2^{n-3} + 2^{n-4} - 3\beta.
\end{aligned}$$

Since $\beta \geq 1$, we conclude the proof by noting that

$$\begin{aligned}
\left\langle 13 \cdot 2^{n-2} \right\rangle_M &= 2^{n-2} + 2^{n-3} + 2^{n-4} - 3\beta < \gamma, \\
\left\langle 14 \cdot 2^{n-2} \right\rangle_M &= 2^{n-1} + 2^{n-3} + 2^{n-4} - 3\beta < \gamma, \text{ and} \\
\left\langle 15 \cdot 2^{n-2} \right\rangle_M &= 2^{n-1} + 2^{n-2} + 2^{n-3} + 2^{n-4} - 3\beta < \gamma.
\end{aligned}$$

B. Width of $P^{(s)}[i]$

Let us prove by induction that $P[i]$ is an $(n+2)$-bit number. Since $P[r] = 0$, we check that $k = 0$, and $T[r-1] = P[r-1] = x_{r-1}Y$, which is an n-bit number. This property holds for $i = r - 1$. Assume now that $P^{(s)}[i+1]$ is an $(n+2)$-bit number. We have to consider two cases according to the parameter α:

- Our hypotheses guarantee that $n_{k-1} \geq 5$ for $\alpha = 2$. Therefore, $2\left\langle P^{(s)}[i+1] \right\rangle_{2^{n-2}}$ contains k sum words of respective widths $n'_0 = n_0 + 1$,..., $n'_{k-2} = n_{k-2}$, and $n'_{k-1} = n_{k-1} - 4$ (Figure 13a). Let us split the partial product x_iY into k blocks in order to add it word by word to $2\left\langle P^{(s)}[i+1] \right\rangle_{2^{n-2}}$. We know that

$$\sum_{i=0}^{k-1} n'_i = n - 1.$$

Since x_iY is an n-bit integer, we deduce from the above equation that its most significant sum word contains $n''_{k-1} = n'_{k-1} + 1 = n_{k-1} - 3$ bits. Therefore the sum of the most significant bits of $2\left\langle P^{(s)}[i+1] \right\rangle_{2^{n-2}}$, x_iY, and a carry bit is bounded by:

$$\begin{aligned}
(2^{n'_{k-1}+1} - 1) + (2^{n'_{k-1}} - 1) + 1 \\
&= 2^{n_{k-1}-3} + 2^{n_{k-1}-4} - 1 \\
&= 3 \cdot 2^{n_{k-1}-4} - 1,
\end{aligned}$$

which is an $(n_{k-1} - 2)$ bit number. Therefore, since $\sum_{i=0}^{k-1} n_i = n + 2$, $T^{(s)}[i]$ is an $(n+1)$-bit number. Indeed, we have:

$$\begin{aligned}
(n_{k-1} - 2) + \sum_{i=1}^{k-2} n_i + (n_0 + 1) &= \sum_{i=0}^{k-1} n_i - 1 \\
&= n + 1.
\end{aligned}$$

Four most significant bits of $P^{(s)}[i+1]$ address the table responsible of the modulo M correction (Figure 13b). Recall that we have to combine the output of this table and carry bits of $T[i]$ in order to generate a high-radix carry-save number $U[i]$, whose format is the one of $P[i]$. Since $2\left\langle k \cdot 2^{n-\alpha} \right\rangle_M$ is an $(n+1)$-bit number, we split it in k words of respective lengths n_0, n_1,..., n_{k-2}, and $(n_{k-1} - 1)$. Consider now the addition of the most significant words of $2\left\langle k \cdot 2^{n-\alpha} \right\rangle_M$ and $T^{(s)}[i]$, and the most significant carry bit of $T^{(c)}[i]$. According to our hypotheses, $n_{k-1} \geq 5$ and this most significant word contains at least 4 bits. Consider the worst case (Figure 13b), where $n_{k-1} = 5$ and the weight of the

most significant bit of $T^{(c)}[i]$ is equal to 2^{n-2}. We deduce from Equation (9) that $U^{(s)}[i]$ is an $(n+2)$-bit number and that its most significant sum word is smaller than or equal to 2^4. The addition of the most significant words of $T^{(s)}[i]$ and $U^{(s)}[i]$, and a carry bit never generates an output carry and $P^{(s)}[i]$ is therefore an $(n+2)$-bit number (Figure 13c).

- Assume now that $\alpha = 1$. The same approach allows us to show that $T^{(s)}[i]$ is an $(n+1)$-bit word (Figure 14a). According to our hypotheses, the most significant word of $P^{(s)}[i-1]$ contains at least six bits. Therefore, the weight of the most significant carry bit of $T^{(c)}[i]$ is at most 2^{n-3}. Since Equation (9) guarantees that $2\left\langle k \cdot 2^{n-\alpha} \right\rangle_M < 2^n + 2^{n-1} + 2^{n-2} + 2^{n-3}$, we deduce that $U^{(s)}[i]$ is an $(n+1)$-bit number (Figure 14b). Note that, for some moduli, we can relax the constraint on n_{k-1}: the remaining of the proof will only assume that $U^{(s)}[i]$ is an $(n+1)$-bit number. An automatic code generator can check this condition very easily for a given value of M. Since the most significant words of $T^{(s)}[i]$ and $U^{(s)}[i]$ have the same size, their addition may generate an output carry and $P^{(s)}[i]$ is therefore an $(n+2)$-bit number (Figure 14c).

C. Final Modulo M Correction

The last step consists in proving that $P[-1]/2$ is smaller than $2M$. We have again to consider two cases according to α:

- Assume that $\alpha = 2$ and consider the last iteration (*i.e.* $i = -1$). Since the partial product $x_{-1}Y$ is equal to zero, we have:

$$\begin{aligned} T[-1] &= 2 \cdot \left(\left\langle P^{(s)}[0] \right\rangle_{2^{n-2}} + P^{(c)}[0] \right) \\ &\leq 2 \cdot \left(2^{n-2} - 1 + 2^{n-2} - 1 \right) = 2^n - 4. \end{aligned}$$

  Thus, $P[-1] \leq 2^n + 2M - 6$ and $P[-1]/2 \leq 2^{n-1} + M - 3$. Since the modulus M is supposed greater than 2^{n-1}, we know that $P[-1]/2$ is smaller than $2M$.
- When $\alpha = 1$, $\left\langle P^{(s)}[i+1] \right\rangle_{2^{n-\alpha}}$ is smaller than or equal to $2^{n-1} - 1$. Recall that the weight of the most significant carry bit of $P^{(c)}[i+1]$ is equal to $n_0 + n_1 + \ldots + n_{k-2}$ (Section II). Thus,

$$T[-1] \leq 2^n - 2 + 2^{n_0+1} + \ldots + 2^{n_0+\ldots+n_{k-2}+1},$$

  and

$$P[-1] \leq 2^n - 2 + 2^{n_0+1} + \ldots + 2^{n_0+\ldots+n_{k-2}+1} + 2\psi_{\text{max}}.$$

  Therefore, $P[-1]/2$ is smaller than $2M$ if

$$\frac{2^{n-1} - 1 + 2^{n_0} + \ldots + 2^{n_0+\ldots+n_{k-2}} + \psi_{\text{max}}}{2} < M.$$

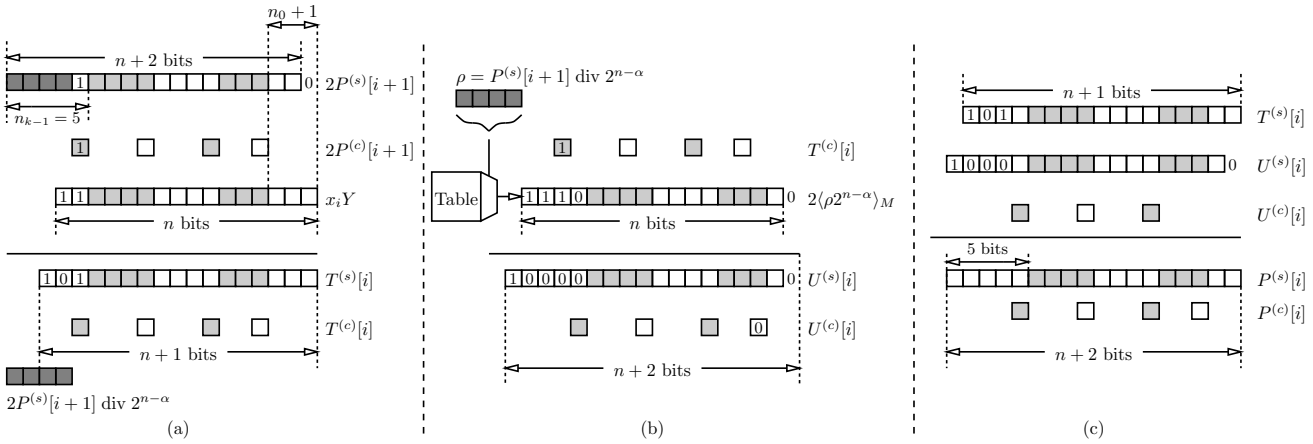


Fig. 13. Proof of Algorithm 1 for $\alpha = 2$. a) Computation of $T[i]$. b) Modulo M reduction and conversion. c) Computation of $P[i]$.

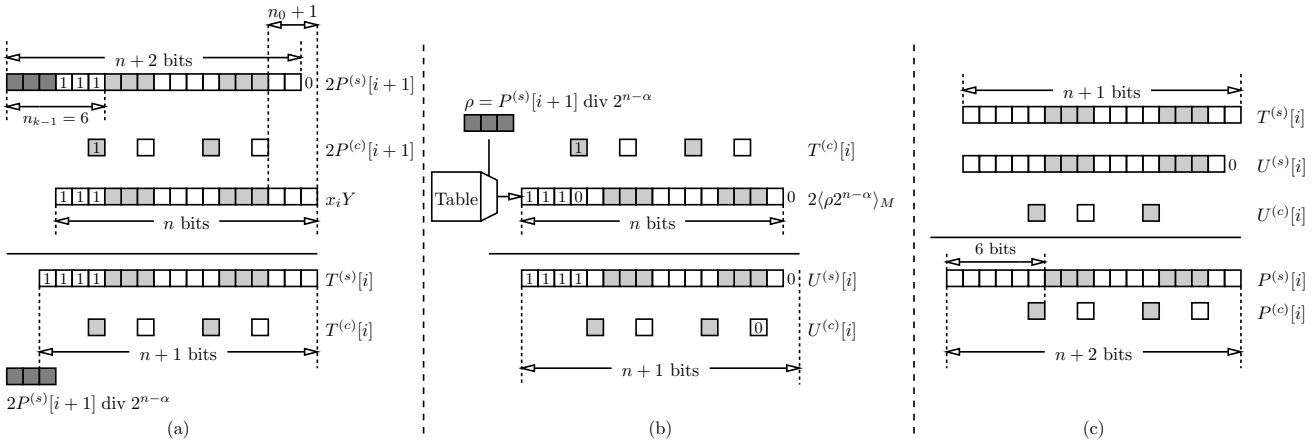


Fig. 14. Proof of Algorithm 1 for $\alpha = 1$. a) Computation of $T[i]$. b) Modulo M reduction and conversion. c) Computation of $P[i]$.