Horner's Rule-Based Multiplication over $\mathbb{F}_p$ and $\mathbb{F}_{p^n}$: A Survey

Jean-Luc Beuchat, Takanori Miyoshi, Jean-Michel Muller, and Eiji Okamoto

LIP Research Report N° 2007–19
LIP/Arénaire, CNRS – INRIA – ENS Lyon – Université Lyon 1

***Abstract*— This paper aims at surveying multipliers based on Horner's rule for finite field arithmetic. We present a generic architecture based on five processing elements and introduce a classification of several algorithms based on our model. We provide the readers with a detailed description of each scheme which should allow them to write a VHDL description or a VHDL code generator.**

***Index Terms*— Modular multiplication, Horner's rule, carry-save, high-radix carry-save, borrow-save, finite field, FPGA.**

I. Introduction

This paper proposes a survey of Horner's rule-based multipliers over $\mathbb{F}_p$ and GF(p^m), where p is a prime number. Multiplication over $\mathbb{F}_p$ is a crucial operation in cryptosystems such as RSA or XTR. Multiplication over GF(p^m) is a fundamental calculation in elliptic curve cryptography, pairing-based cryptography, and implementation of error-correcting codes.

In the following, the modulus F is either an n-bit (prime) integer whose most significant bit is set to one (i.e. $2^{n-1}+1 \leq F \leq 2^n-1$) or a monic degree-$n$ irreducible polynomial over $\mathbb{F}_p$. Three families of algorithms allow one to compute the product AB modulo F, where A and B are either elements of $\mathbb{Z}/F\mathbb{Z}$ or $\mathbb{F}_{p^n}$. In parallel-serial schemes, a single digit or coefficient of the multiplier A is processed at each step. This leads to small operands performing a multiplication in n clock cycles. Parallel multipliers compute the product AB ($2n$-bit integer or degree-$(2n-2)$ polynomial) and carry out a final modular reduction. They achieve a higher throughput at the price of a larger circuit area. Song and Parhi introduced array multipliers as a trade-off between computation time and circuit area [1]. Their idea consists in processing D digits or coefficients of the multiplier at each step. The parameter D is sometimes referred to as *digit size* and parallel-serial schemes can be considered as a special case with $D=1$. In such architectures, the multiplier A can be processed starting with the least significant element (LSE) or the most significant element (MSE). This survey is devoted to MSE operators and we refer the reader to [2], [3], [4] for details about parallel modular multipliers and LSE operators, which are often based on the celebrated Montgomery algorithm [5]. Note that Kaihara and Takagi introduced a novel representation of residues modulo F which allows the splitting of the multiplier A [6]: its upper and lower parts are processed independently using an MSE scheme and an LSE implementation of the Montgomery algorithm respectively. Such an approach could potentially divide the computation time of array multipliers by two.

After a brief description of the five number systems considered in this survey (Section II), we outline the architecture of a modular multiplier based on Horner's rule (Section III). We then introduce a classification of several MSE schemes according to our model, and provide the reader with all the details needed for writing a VHDL description or designing a VHDL code generator (Sections IV, V, and VI). We conclude this survey by a comparison of the most promising algorithms on a typical field-programmable gate array (FPGA) architecture (Section VII).

J.-L. Beuchat, T. Miyoshi, and E. Okamoto are with the University of Tsukuba, Tsukuba, Japan. J.-M. Muller is with the *CNRS, laboratoire LIP, projet Arénaire*, Lyon, France.

II. Number Systems

This section describes the number systems involved in the algorithms we survey in this paper. We also outline addition algorithms and describe how to compute a number or polynomial $\tilde{A}$ congruent to A modulo F.

A. Radix-2 Integers

1) Addition of Radix-2 Integers: Let A and B be two n-bit unsigned integers such that $A=\sum_{i=0}^{n-1} a_i 2^i$ and $B=\sum_{i=0}^{n-1} b_i 2^i$. A carry-ripple adder (CRA), whose basic building blocks are the full-adder (FA) and the half-adder (HA) cells, returns the $(n+1)$-bit sum $R=A+B$ (Figure 1). Since a CRA consists of a linearly connected array of FAs, its delay grows linearly with n, thus making this architecture inadvisable for an ASIC implementation of high-speed applications. Modern FPGAs being mainly designed for digital signal processing applications involving rather small operands (16 up to 32 bits), manufacturers chose to embed dedicated carry logic allowing the implementation of fast CRAs for such operand sizes. The design of modular multipliers taking advantage of such resources is therefore of interest. An application would for instance be the FPGA implementation of the Montgomery modular multiplication algorithm in a residue number system [7].

2) Modular Reduction: Modulo F reduction can be implemented by means of comparisons and subtractions. It is sometimes easier to compute an $(n+1)$-bit number $\tilde{A}$ congruent to an $(n+q)$-bit number A modulo F. Let us define $A_{k:j} = \sum_{i=j}^{k} a_i 2^{i-j}$, where $k \geq j$. Using this notation, A is equal to $A_{n+q-1:n}2^n + A_{n-1:0}$. If q is small enough, we can store in a table all values of $(A_{n+q-1:n}2^n) \bmod F$ and compute $\tilde{A}$ by means of a single CRA: $\tilde{A} = (A_{n+q-1:n}2^n) \bmod F + A_{n-1:0}$.

Note that some algorithms studied in this survey also involve negative integers. We encode such numbers using the two's complement system. An n-bit number $A \in \{-2^{n-1}, \ldots, 2^{n-1}-1\}$ is represented by $A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$.

B. Carry-Save Numbers

1) Addition of Carry-Save Numbers: Figure 1b describes a carry-save adder (CSA). This operator computes in constant time the sum of three n-bit operands by means of n FAs. It returns two n-bit numbers $R^{(s)}$ and $R^{(c)}$ containing the sum and output carry bits of the FAs respectively. We have:

$$\begin{aligned} R &= 2R^{(c)} + R^{(s)} \\ &= r_0^{(s)} + \sum_{i=1}^{n-1} \left(r_i^{(s)} + r_{i-1}^{(c)}\right) 2^i + r_{n-1}^{(c)} 2^n \\ &= \sum_{i=0}^{n} r_i 2^i, \end{aligned}$$

where $r_0 = r_0^{(s)}$, $r_n = r_{n-1}^{(c)}$, and $r_i = r_i^{(s)} + r_{i-1}^{(c)}$, $1 \leq i \leq n-1$. Each digit r_i belonging to $\{0,1,2\}$, we obtain a radix-2 redundant number system. Unfortunately, comparison and modular reduction require a carry propagation and we would lose all benefits from this number system by introducing such operations in modular multiplication algorithms.

2) Modular Reduction: Let A be an n-bit two's complement number whose carry-save representation is given by $A = A^{(s)} + 2A^{(c)}$. Koç and Hung introduced a sign estimation technique which enables computing a number congruent to A modulo F by inspecting a few most significant bits of $A^{(s)}$ and $A^{(c)}$ [8], [9], [10]. They define the truncation function $\Theta(A)$ as the operation which replaces the least significant τ bit of A with zeroes. The parameter τ control the cost and the quality of the estimation. Let k be the two's complement sum of $\Theta(A^{(s)}) + \Theta(2A^{(c)})$. The sign estimation function $\mathrm{ES}(A^{(s)}, A^{(c)})$ is then defined as follows [10]:

$$\mathrm{ES}(A^{(s)}, A^{(c)}) = \begin{cases} (+) & \text{if } k \geq 0, \\ (-) & \text{if } k < -2^{\tau}, \\ (\pm) & \text{otherwise.} \end{cases}$$

Koç and Hung proved that, if $\mathrm{ES}(A^{(s)}, A^{(c)}) = (+)$ or $(-)$, then $X \geq 0$ or $X < 0$, respectively [10]. If $\mathrm{ES}(A^{(s)}, A^{(c)}) = (\pm)$, then $-2^{\tau} \leq A < 2^{\tau}$. One can therefore add $-F$, 0, or F to A according to the result of the sign estimation to compute a number $\tilde{A}$ congruent to A modulo F.

3) Modular Reduction when the Modulus is a Constant: Assume now that the n-bit modulus F is known at design time and consider a carry-save number A such that $A^{(s)}$ and $A^{(c)}$ are n_s- and n_c-bit integers respectively (n_s and n_c are usually greater than or equal to n). Let $\alpha \leq n$. Since

$$\begin{aligned} A &= \left(A^{(s)} \text{ div } 2^{\alpha} + (2A^{(c)}) \text{ div } 2^{\alpha}\right) \cdot 2^{\alpha} + \\ &\quad A^{(s)} \bmod 2^{\alpha} + (2A^{(c)}) \bmod 2^{\alpha} \\ &= \left(A^{(s)}_{n_s-1:\alpha} + A^{(c)}_{n_c-1:\alpha-1}\right) \cdot 2^{\alpha} + A^{(s)}_{\alpha-1:0} + 2A^{(c)}_{\alpha-2:0}, \end{aligned}$$

we compute a number $\tilde{A}$ congruent to A by means of a CSA and a table addressed by $\max(n_s + 1 - \alpha, n_c + 2 - \alpha)$ bits. Let $k' = (A^{(s)}_{n_s-1:\alpha} + A^{(c)}_{n_c-1:\alpha-1}) \cdot 2^{\alpha}$. We have:

$$A \equiv k' \bmod F + A^{(s)}_{\alpha-1:0} + 2A^{(c)}_{\alpha-2:0} \pmod F. \quad (1)$$

We easily compute an upper bound for $\tilde{A}$. Since $k \bmod F \leq F - 1$, we have:

$$\tilde{A} \leq F - 1 + 2^{\alpha} - 1 + 2(2^{\alpha-1} - 1) = F + 2^{\alpha+1} - 4. \quad (2)$$

C. High-Radix Carry-Save Numbers

Carry-save adders do not always take advantage of the dedicated carry logic available in modern FPGAs [11]. To overcome this problem, modular multiplication can be performed in a high-radix carry-save number system, where a sum bit of the carry-save representation is replaced by a sum word. A q-digit high-radix carry-save number A is denoted by

$$A = (a_{q-1}, \ldots, a_0) = \left(\left(a_{q-1}^{(c)}, a_{q-1}^{(s)}\right), \ldots, \left(a_0^{(c)}, a_0^{(s)}\right)\right),$$

where the jth digit a_j consists of an n_j-bit sum word $a_j^{(s)}$ and a carry bit $a_j^{(c)}$ such that $a_j = a_j^{(s)} + a_j^{(c)} 2^{n_j}$. Let us define $A^{(s)} = a_0^{(s)} + a_1^{(s)} 2^{n_0} + \ldots + a_{q-1}^{(s)} 2^{n_0+\ldots+n_{q-2}}$ and $A^{(c)} = a_0^{(c)} 2^{n_0} + a_1^{(c)} 2^{n_0+n_1} + \ldots + a_{q-1}^{(c)} 2^{n_0+\ldots+n_{q-1}}$. With this notation, a number A is equal to $A^{(s)} + A^{(c)}$. This number system has nice properties to deal with large numbers on FPGAs:

- Its redundancy allows one to perform addition in constant time (the critical path of a high-radix carry-save adder only depends on $\max_{0 \leq j \leq q-1} n_j$).
- The addition of a sum word $a_j^{(s)}$, a carry bit $a_{j-1}^{(c)}$, and an n_j-bit unsigned binary number is performed by means of a CRA.

Unfortunately, MSE first algorithms involve left-shift operations which modify the representation of an operand. Figure 2 describes a 4-digit high-radix carry-save number $A = 2260$ with $n_0 = n_1 = 3$, $n_2 = 4$, and $n_3 = 3$. By shifting A, we obtain $B = 2A$, whose least significant sum word is now a 4-bit number.

D. Borrow-Save Numbers

1) Addition of Borrow-Save Numbers: A radix-r signed-digit representation of a number $A \in \mathbb{Z}$ is given by $A = \sum_{i=0}^{n} a_i r^i$. The digits a_i belong to $\mathcal{D}_r = \{-\rho, -\rho+1, \ldots, \rho -$

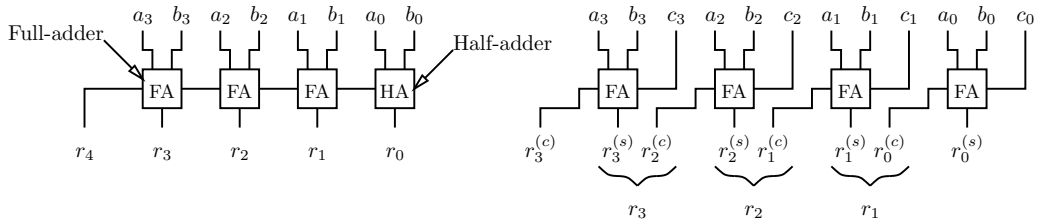


Fig. 1. Carry-ripple adder and carry-save adder.

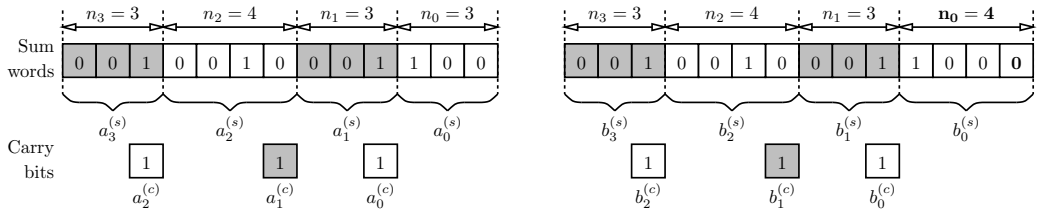


Fig. 2. High-radix carry-save numbers.

$1, \rho\}$, where $\rho \leq r-1$ and $2\rho+1 \geq r$. The second condition guarantees that every number has a representation ($2\rho+1 = r$). When $2\rho+1 > r$, the number system becomes redundant and allows one to perform addition in constant time under certain conditions [12].

In this survey, we will consider only radix-2 signed-digits. Thus, we take advantage of the borrow-save notation introduced by Bajard *et al.*[13]: each digit a_i is encoded by a positive bit a_i^+ and a negative bit a_i^- such that $a_i = a_i^+ - a_i^-$. A modified FA cell, called PPM cell, allows one to compute two bits r_{i+1}^+ and r_i^- such that $2r_{i+1}^+ - r_i^- = a_i^+ + b_i^+ - a_i^-$. Note that the same cell is also able to return r_{i+1}^- and r_i^+ such that $2r_{i+1}^- - r_i^+ = a_i^- + b_i^- - a_i^+$. In this case, it is usually referred to as MMP cell. The addition of two borrow-save numbers can be performed in constant time using the operator described by Figure 3a [13].

2) Modular Reduction: Assume that A is an $(n+2)$-digit borrow-save number such that $-2F < A < 2F$. Takagi and Yajima proposed a constant time algorithm which returns an $(n+1)$-digit number $\tilde{A}$ congruent to A modulo F (Figure 3b) [14]. First, we add the three most significant digits of A and get a 4-bit two's complement number $k = 4a_{n+1} + 2a_n + a_{n-1}$. Our hypotheses guarantee that $-4 \leq k \leq 4$ and

$$\begin{aligned} -2F < A < 0, \text{ if } k < 0, \\ -2^{n-1} < A < 2^{n-1}, \text{ if } k = 0, \text{ and} \\ 0 < A < 2F, \text{ if } k > 0. \end{aligned}$$

Thus, it suffices to add F, 0, or $-F$ to A according to k in order to get an $(n+1)$-digit number $\tilde{A}$ such that $-F < A < F$. Since we assumed that the most significant bit of F is always set to one, we have

$$\begin{aligned} -F &= -2^{n-1} - \sum_{i=0}^{n-2} f_i 2^i \\ &= -2^n + 2^{n-1} - \sum_{i=0}^{n-2} f_i 2^i \\ &= -2^n + \sum_{i=0}^{n-2} (1-f_i) 2^i + 1. \end{aligned}$$

Consider now the $(n+1)$-digit borrow-save number U defined as follows:

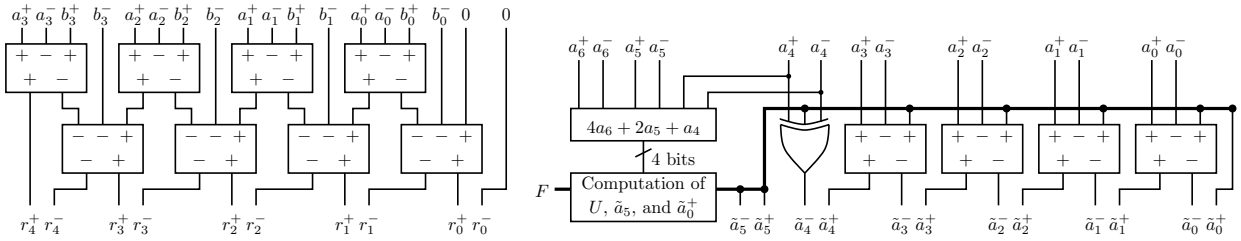
$$U = \begin{cases} F = \sum_{i=0}^{n-1} f_i 2^i & \text{if } k < 0, \\ 0 & \text{if } k = 0, \\ -F - 1 = -2^n + \sum_{i=0}^{n-2} (1-f_i) 2^i & \text{if } k > 0, \end{cases}$$

and note that most significant digit u_n is the only one which can take a negative value. The $(n+1)$-digit sum $\tilde{A} = A + U$ can therefore be computed by a single stage of PPM cells and glue logic (Figure 3b). Since $U = -F - 1$ when k is greater than 0, a small table generates $\tilde{a}_0^+$ according to the following rule:

$$\tilde{a}_0^+ = \begin{cases} 1 & \text{if } k > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Consider now the addition of a_{n-1}^+, a_{n-1}^-, and u_{n-1} by means of a PPM cell. It generates two bits v and $\tilde{a}_{n-1}^-$ such that $2v - \tilde{a}_{n-1}^- = a_{n-1}^+ - a_{n-1}^- + u_{n-1}$. The most significant digit $\tilde{a}_n$ is then defined as follows:

$$\tilde{a}_n = \begin{cases} 2a_{n+1} + a_n + v - 1 & \text{if } k > 0, \\ 2a_{n+1} + a_n + v & \text{otherwise.} \end{cases}$$



a) Borrow-save adder

b) Computation of a number $\tilde{A}$ congruent with A modulo F ($n = 5$)

Fig. 3. Arithmetic operations in the borrow-save number system.

Thus, $\tilde{a}_n$ only depends on k. Instead of explicitly computing v, we build a table addressed by a_{n+1}, a_n, and a_{n-1} (Table I).

TABLE I
COMPUTATION OF THE MOST SIGNIFICANT DIGIT OF $\tilde{A}$.

a_{n+1}	a_n	a_{n-1}	$\tilde{a}_n$
0	0	0	0
0	0	1	0
0	1	−1	0
1	−1	−1	0
0	1	0	0
1	−1	0	0
0	1	1	1
1	0	−1	1
1	−1	1	1
1	0	0	1

a_{n+1}	a_n	a_{n-1}	$\tilde{a}_n$
0	0	−1	0
0	−1	1	0
−1	1	1	0
0	−1	0	0
−1	1	0	0
0	−1	1	−1
−1	0	−1	−1
−1	1	1	−1
−1	0	0	−1

E. Elements of $\mathbb{F}_{p^n}$

There are several ways to encode elements of an extension field. In this paper, we will only consider the well-known polynomial representation, which is for instance often faster than normal basis in pairing-based applications [15]. Let $F(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1 x + f_0$ be an irreducible polynomial over $\mathbb{F}_p$, where p is a prime. Then, GF(p^n) = GF(p)[x]/$F(x)$, and an element $a(x) \in$ GF(p^n) can be represented by a degree-$(m-1)$ polynomial with coefficients in $\mathbb{F}_p$: $a(x) = a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$.

Note that the irreducible polynomials used in cryptographic applications are commonly binomials or trinomials, thus making modulo F operations easy to implement. $F = x^{97}+x^{12}+2$ is for instance irreducible over GF(3). Assume that A is a degree-97 polynomial. It suffices to remove $a_{97} \cdot F = a_{97}x^{97} + a_{97}x^{12} + 2a_{97}$ from A to get $A \mod F$ and this operation involves only two multiplications and two subtractions over GF(3), namely $a_{12} - 1 \cdot a_{97}$ and $a_0 - 2 \cdot a_{97}$. Elements of GF(3) are usually encoded with two bits and such a modular reduction is performed by means of two 4-input tables.

III. HORNER'S RULE FOR MODULAR MULTIPLICATION

Recall that the celebrated Horner's rule suggests to compute the product of two n-bit integers or degree-$(n-1)$ polynomials A and B as follows:

$$AB = (\ldots((a_{n-1}B)^{\ll 1} + a_{n-2}B)^{\ll 1} + \ldots)^{\ll 1} + a_0 B,$$

where $^{\ll 1}$ denotes the left-shift operation (i.e. multiplication by two for integers and multiplication by x for polynomials). This scheme can be expressed recursively as follows:

$$R[i] = R[i+1]^{\ll 1} + a_i B, \tag{3}$$

where the loop index i goes from $n-1$ to 0, $R[n] = 0$, and $R[0] = AB$. By performing a modular addition at each step, one easily determines the product $AB \mod F$ [16]. However, computing a number or polynomial $R[i]$ congruent to $R[i+1]^{\ll 1} + a_i B$ modulo F by inspecting a few most significant digits or coefficients may lead to a smaller and faster iteration stage. Thus, besides a register storing the intermediate results, a modular multiplier based on Horner's rule consists of four blocks (Figure 4):

- A partial product generator (PPG), whose architecture depends on the number system, computes $a_i B$. If operands are radix-2 integers, a PPG comprises for instance n AND gates. For elements of $\mathbb{F}_{p^n}$, a PPG consists of n modulo p multipliers. Since p is generally equal to two or three in elliptic curve or pairing-based cryptography, these modulo p multiplications are efficiently implemented by means of AND gates or small tables.
- The *Modshift* block computes a number (or a polynomial) $S[i]$ which is congruent to $R[i+1]^{\ll 1}$ modulo F. We suggest in this survey a classification of Horner's rule-based modular multipliers according to the architecture of this block. Since

$$\begin{aligned} R[i+1]^{\ll 1} &\equiv R[i+1]^{\ll 1} - k_1 F \pmod F \\ &\equiv (R[i+1] - k_2 F)^{\ll 1} \pmod F, \end{aligned}$$

  where k_1 and k_2 are integers or polynomials, we consider three families of algorithms. In *left-shift* schemes, $S[i]$ is equal to $R[i+1]^{\ll 1}$. The *left-shift and modular reduction* approach returns a number or polynomial $S[i]$ congruent to $R[i+1]^{\ll 1}$ modulo F. Finally, the *modular reduction and left-shift* solution consists in left-shifting a number or polynomial congruent to $R[i+1]$ modulo F. Table II summarizes the algorithms we consider in this survey according to this model.
- The *Modsum* module is responsible for the computation of a new intermediate result $R[i]$ congruent to $S[i] + a_i B$ modulo F.
- A final modular reduction is required when the algorithm returns a number (or a polynomial) congruent to $AB \mod F$. Furthermore, a conversion from a redundant number system to a standard radix-2 representation is sometimes required. The *Modred* module performs these tasks.

TABLE II
CLASSIFICATION OF MODULAR MULTIPLIERS BASED ON HORNER'S RULE ACCORDING TO THE ARCHITECTURE OF THE *Modshift* BLOCK.

	Left-shift and modular reduction	**Left-shift**	**Modular reduction left-shift**
Borrow-save	Takagi and Yajima [14] Takagi [17]		
Carry-save	Jeong and Burleson [18]	Koç and Hung [9] Koç and Hung [10] Kim and Sobelman [21] Amanor *et al.* [22]	Bunimov and Schimmler [19] Peeters *et al.* [20]
High-radix carry-save			Beuchat and Muller [11]
Radix 2		Beuchat and Muller[23]	
$\mathbb{F}_{p^n}$	Shu *et al.* [24]		Song and Parhi [1]

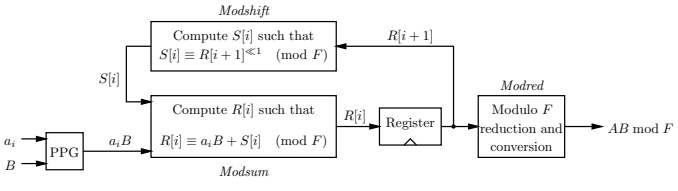


Fig. 4. Modular multiplication based on Horner's rule.

IV. FIRST ARCHITECTURE: LEFT-SHIFT OPERATION FOLLOWED BY A MODULAR REDUCTION

A. Borrow-Save Algorithms

Let A and B be two $(n+1)$-digit borrow-save numbers with $-F < A, B < F$. Takagi and Yajima proposed an algorithm computing an $(n+1)$-digit number $R[0] \in \{-F+1, \ldots, F-1\}$ congruent to AB modulo F [14] (Figure 5). At each step, the *Modshift* block returns an $(n+1)$-digit number $S[i] \in \{-F+1, \ldots, F-1\}$ congruent to the $(n+2)$-digit number $2R[i+1]$ according to the scheme described in Section II-D. The *Modsum* block contains a borrow-save adder which computes the sum $T[i] \in \{-2F+1, \ldots, 2F-1\}$ of $S[i]$ and a partial product $a_i B$. The same approach allows one to determine a number $R[i] \in \{-F+1, \ldots, F-1\}$ congruent to $T[i]$ modulo F. A nice property of this algorithm is that both inputs and output belong to $\{-F+1, \ldots, F-1\}$. The conversion from borrow-save to integer involves at most two additions:

$$R = AB \bmod F = \begin{cases} R^+[0] - R^-[0] & \text{if } R^+[0] - R^-[0] \geq 0, \\ R^+[0] - R^-[0] + F & \text{otherwise,} \end{cases}$$

where $R^+[0] = \sum_{i=0}^{n-1} r_i^+[0] 2^i$ and $R^-[0] = \sum_{i=0}^{n-1} r_i^-[0] 2^i$. The number of iterations can be reduced by considering a higher radix. Radix-4 modular multipliers based on signed-digits are for instance described in [14], [17].

B. Carry-Save Algorithm

Jeong and Burleson described a carry-save implementation of the algorithm by Takagi and Yajima [14] in the case where the modulus F is known at design time [18] (Figure 6). The intermediate result $R[i]$ is represented by two n-bit unsigned integers $R^{(s)}[i]$ and $R^{(c)}[i]$. The *Modshift* block implements Equation (1) and returns a carry-save number $S[i]$ congruent to $2R[i+1]$, while the *Modsum* block requires two CSAs to determine a number $R[i]$ congruent to $S[i] + a_i B$. According to Equation (2), $R[i]$ is smaller than or equal to $F + 2^{n+1} - 4$ and the *Modred* block has to remove up to $4F$ to $R[0]$ in order to get $AB \bmod F$.

Kim and Sobelman proposed an architecture based on four fast adders (e.g. carry-select adders or parallel-prefix adders) to perform this final modular reduction and to convert the result from carry-save to integer [21] (Figure 7). They first compute an $(n+1)$-bit integer U such that $U = R^{(s)}[0] + 2R^{(c)}_{n-2:0}$. Then, a second adder and a table addressed by $r^{(c)}_{n-1}[0]$ and u_n return an $(n+1)$-bit integer $V = U_{n-1:0} + ((r^{(c)}_{n-1}[0] + u_n) \cdot 2^n) \bmod F$. Since $V \leq 2^n + F - 2 < 3F$, it suffices to compute in parallel $V - 2F$ and $V - F$, and to select the result.

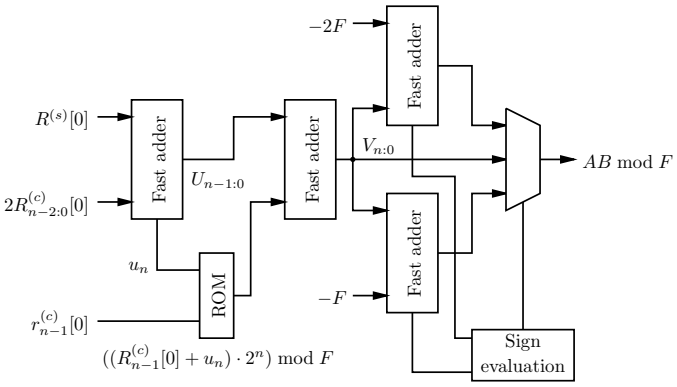


Fig. 7. Architecture of the *Modred* block proposed by Kim and Sobelman [21].

C. Multiplication over $\mathbb{F}_{p^n}$

Shu *et al.* designed an array multiplier processing D coefficients of the operand A at each clock cycle [24] (Figure 8a). The intermediate result $R[i]$ is a degree-$(n-1)$ polynomial, thus avoiding the need for a final modular reduction. At each step, the *Modshift* block returns a degree-$(n-1)$ polynomial $S[i]$ equal to $x^D R[i+1] \bmod F$. A $(D+1)$-operand adder computes the sum of $S[i]$ and D partial products reduced

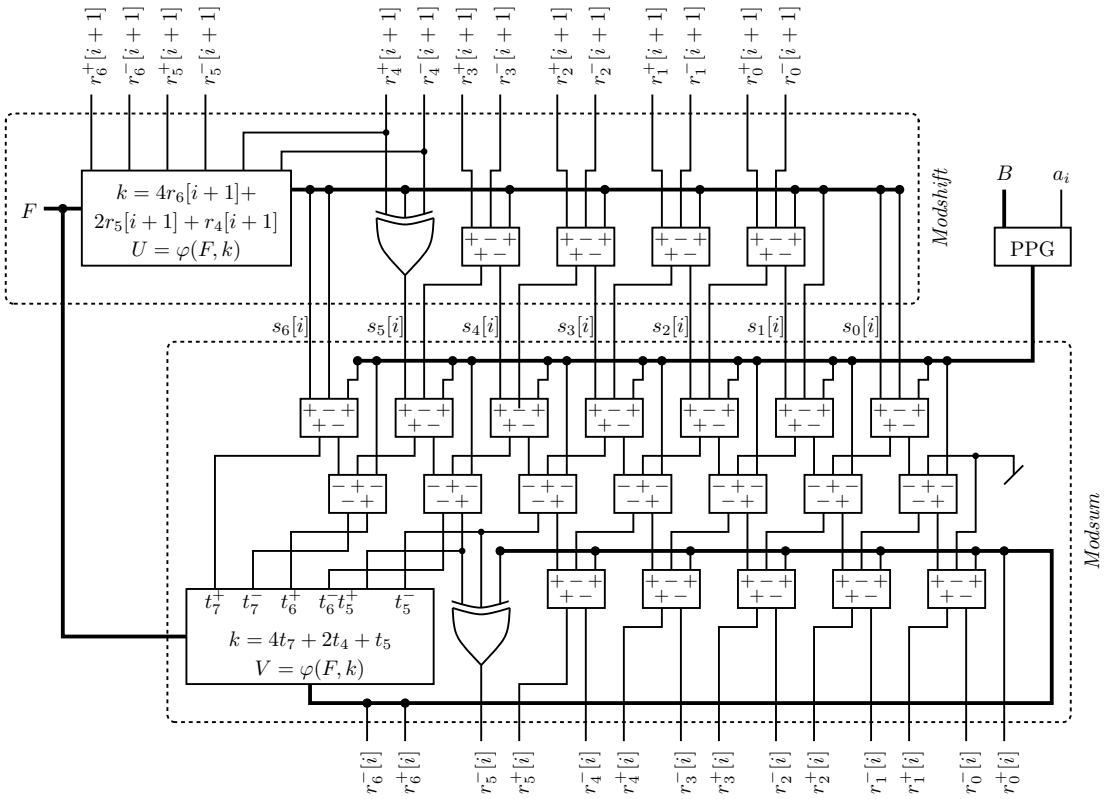


Fig. 5. Architecture of the iteration stage proposed by Takagi and Yajima [14] for $n = 6$.

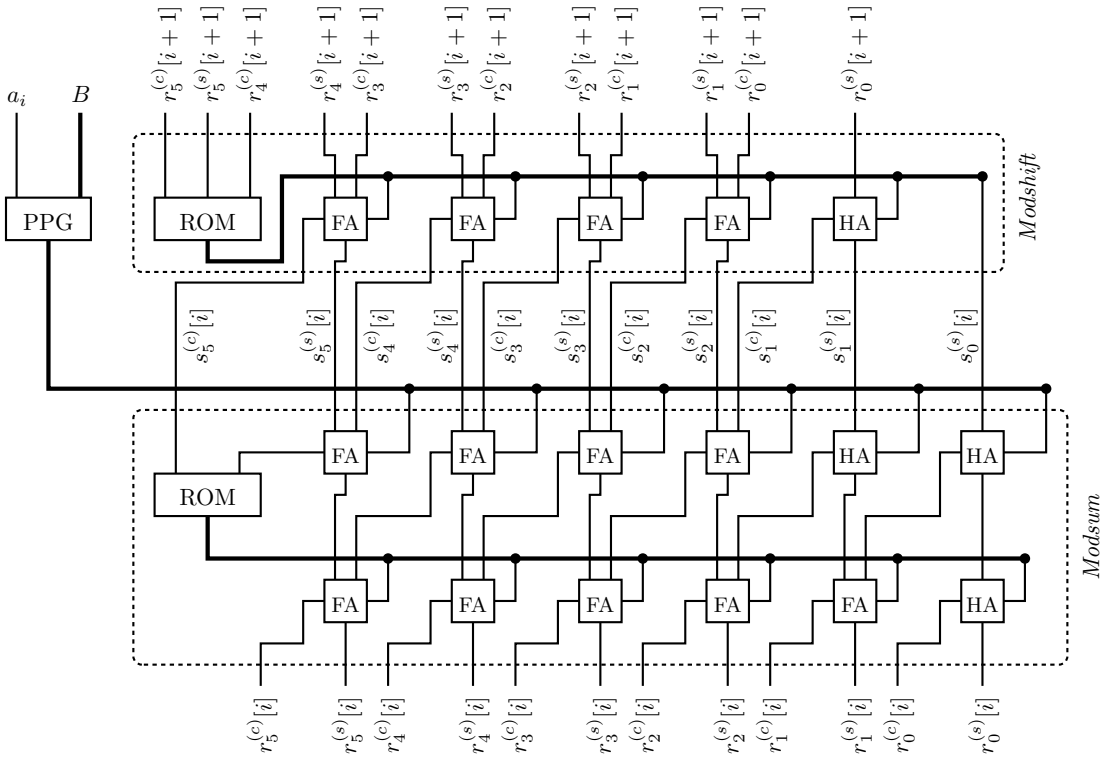


Fig. 6. Architecture of the iteration stage proposed by Jeong and Burleson [18] for $n = 6$.

modulo F: $R[i] = S[i] + \sum_{j=0}^{D-1} \big((x^j a_{Di+j} B) \bmod F\big)$.

V. Second Architecture: Left-Shift Operation

A. Carry-Save Algorithms

1) First Case: the Modulus is an Input of the Operator: Koç and Hung designed a modular multiplier based on their

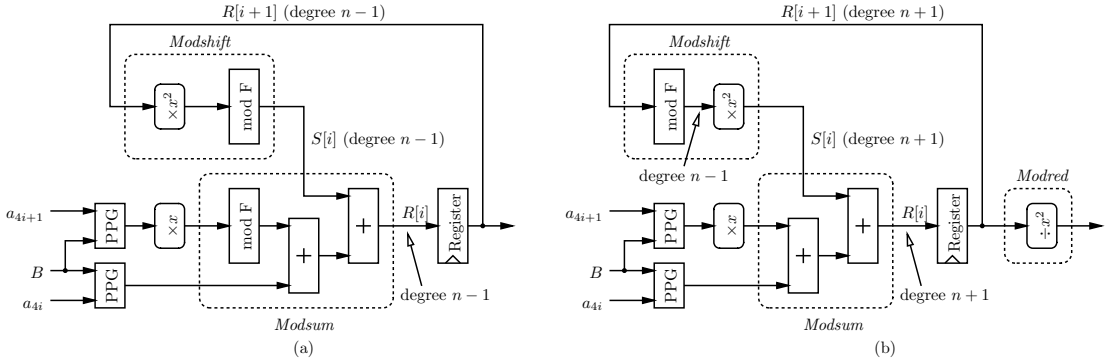


Fig. 8. Array multipliers over $GF(p^n)$ processing $D = 2$ coefficients of A at each clock cycle. (a) Architecture proposed by Shu *et al.* [24]. (b) Architecture introduced by Song and Parhi [1].

sign estimation technique outlined in Section II-B [10]. They chose the parameter $\tau = n - 1$ to control the quality of the estimation and introduced a slightly different function defined as follows:

$$\text{ES'}(R^{(s)}[i+1], R^{(c)}[i+1]) = \begin{cases} (+) & \text{if } k \geq 2^n, \\ (-) & \text{if } k < -2^{n+1}, \\ (\pm) & \text{otherwise,} \end{cases} \quad (4)$$

where $R^{(s)}[i+1]$ and $R^{(c)}[i+1]$ are $(n+4)$- and $(n+3)$-bit two's complement numbers respectively. The two's complement number k is therefore computed as follows: $k = R^{(s)}_{n+3:n-1}[i+1] + R^{(c)}_{n+2:n-2}[i+1]$. Koç and Hung established that all intermediate results of their algorithm belong to $\{-6F, -6F+1, \ldots, 7F-1, 7F\}$. Thus the computation of k does not generate an output carry and k is a 5-bit two's complement number. At each step, the *Modsum* block computes $R[i]$ such that

$$R^{(s)}[i]+2R^{(c)}[i] = \begin{cases} 2R^{(s)}[i+1] + 4R^{(c)}[i+1] + a_i B - 8F \\ \quad \text{if ES'}(k) = (+), \\ 2R^{(s)}[i+1] + 4R^{(c)}[i+1] + a_i B + 8F \\ \quad \text{if ES'}(k) = (-), \\ 2R^{(s)}[i+1] + 4R^{(c)}[i+1] + a_i B \\ \quad \text{otherwise.} \end{cases}$$

After n clock cycles, we get $R[0] = AB + 8\alpha F$, with $\alpha \in \mathbb{Z}$. Koç and Hung suggested to perform three additional iterations with $a_{-1} = a_{-2} = a_{-3} = 0$ in order to obtain $R[-3] = 8AB + 8\beta F \in \{-6F, \ldots, 7F\}$, with $\beta \in \mathbb{Z}$. Since $R[-3]$ is a multiple of eight, a right-shift operation returns a number R congruent to AB modulo F, where $-F < R < F$. After the conversion to two's complement, the *Modred* module has to perform at most one addition.

Figure 9 describes the iteration stage. We propose here an improved architecture which is based on the following observation: $r^{(s)}_{n+3}[i]$, $r^{(c)}_{n+2}[i]$, and $r^{(s)}_{n+2}[i]$ only depend on k. We can therefore compute these bits while performing the sign estimation (recall that the same idea was exploited for the design of the borrow-save operator introduced by Takagi and Yajima [14] (Section IV-A)). The first step consists in computing the sum $T[i]$ of a partial product $a_i B$ and $2R[i+1]$. Note that $r^{(c)}_0[i+1]$ is always equal to zero. Thus, the adder consists of a 5-bit CRA and an $(n-1)$-input CSA ($n-3$ FAs and 2 HAs):

$$\begin{aligned} T^{(s)}_{n+4:n}[i] &= R^{(s)}_{n+3:n-1}[i+1] + \\ &\quad R^{(c)}_{n+2:n-2}[i+1] = k, \\ T^{(s)}_{n-1:1}[i] + 2T^{(c)}_{n-1:1}[i] &= R^{(s)}_{n-2:0}[i+1] + \\ &\quad 2R^{(c)}_{n-3:0}[i+1] + a_i B_{n-1:1}, \\ t^{(s)}_0[i] &= a_i b_0. \end{aligned}$$

The sign estimation defined by Equation (4) is then computed as follows:

$$\text{ES'}(R^{(s)}[i+1], R^{(c)}[i+1]) = \begin{cases} (+) & \text{if } \bar{k}_4(k_3 + k_2 + k_1) = 1, \\ (-) & \text{if } k_4(\bar{k}_3 + \bar{k}_2 + \bar{k}_1\bar{k}_0) = 1, \\ (\pm) & \text{otherwise.} \end{cases}$$

These logic equations can be computed using Karnaugh maps. Let us define where $\text{es}_+ = \bar{k}_4(k_3 + k_2 + k_1) = 1$ and $\text{es}_- = k_4(\bar{k}_3 + \bar{k}_2 + \bar{k}_1\bar{k}_0)$. If the sign estimation block returns $(+)$ (i.e. $\text{es}_+ = 1$), we have to subtract $8F$ from $T[i]$. Recall that the most significant bit of F is always set to one. Therefore, $-8F - 1$ is encoded by an $(n+4)$-bit two's complement number $(10\underbrace{\bar{f}_{m-2}\bar{f}_{m-3}\ldots\bar{f}_1\bar{f}_0}_{n-1 \text{ bits}}111)_2$. We suggest to represent $-8F$ as follows (Figure 9):

$$-8F = (10\underbrace{\bar{f}_{m-2}\bar{f}_{m-3}\ldots\bar{f}_1\bar{f}_0}_{n-1 \text{ bits}}000)_2 + 2^2\text{es}_+ + 2(\text{es}_+ + \text{es}_+).$$

Finally, Table III summarizes the logic equations defining $r^{(c)}_{n+2}[i]$, $r^{(s)}_{n+3}[i]$, and $r^{(s)}_{n+2}[i]$.

2) Second Case: the Modulus is a Constant: If the modulus is known at design time, an architecture introduced by Kim and Sobelman [21] allows one to replace the sign estimation unit with a table addressed by four bits (Figure 10). The authors suggest to compute a first carry-save number $T[i]$ such that

$$T^{(s)}_{n-1:0}[i] + 2T^{(c)}_{n-1:0}[i] = a_i B + 2R^{(s)}_{n-2:0}[i+1] + 4R^{(c)}_{n-3:0}[i+1].$$

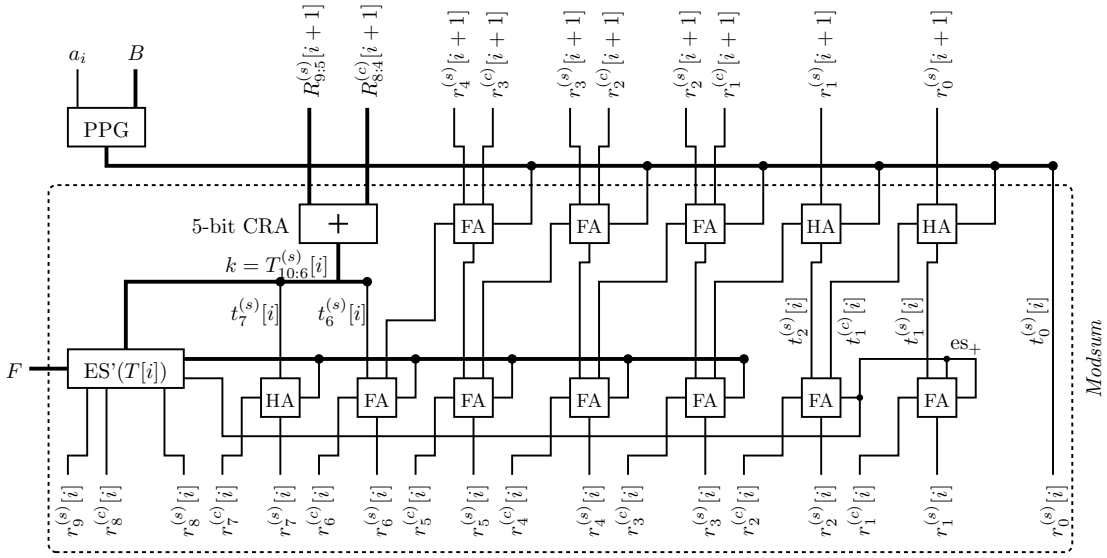


Fig. 9. Architecture of the iteration stage proposed by Koç and Hung [10] for $n = 6$.

TABLE III
ITERATION STAGE PROPOSED BY KOÇ AND HUNG [10]: COMPUTATION OF $r^{(c)}_{n+2}[i]$, $r^{(s)}_{n+3}[i]$, AND $r^{(s)}_{n+2}[i]$.

es$_+$	es$_-$	$r^{(c)}_{n+2}$	$r^{(s)}_{n+3}$	$r^{(s)}_{n+2}$
1	0	0	$\bar{t}^{(s)}_{n+3}$	$t^{(s)}_{n+2}$
0	1	$\bar{t}^{(s)}_{n+3} + \bar{t}^{(s)}_{n+2}$	$\bar{t}^{(s)}_{n+3}\bar{t}^{(s)}_{n+2}$	$\bar{t}^{(s)}_{n+2}$
0	0	0	$t^{(s)}_{n+3}$	$t^{(s)}_{n+2}$

Thus,

$$\begin{aligned} 2R[i+1] + a_iB &= T^{(s)}_{n-1:0}[i] + 2T^{(c)}_{n-1:0}[i] + \\ &\quad \Big(R^{(s)}_{n-1}[i+1] + \\ &\quad R^{(c)}_{n-1:n-2}[i+1]\Big) \cdot 2^n \\ &= T^{(s)}_{n-1:0}[i] + 2T^{(c)}_{n-2:0}[i] + \\ &\quad \Big(R^{(s)}_{n-1}[i+1] + R^{(c)}_{n-1:n-2}[i+1] + \\ &\quad t^{(c)}_{n-1}[i]\Big) \cdot 2^n \end{aligned}$$

Let $k = R^{(s)}_{n-1}[i+1] + R^{(c)}_{n-1:n-2}[i+1] + T^{(c)}_{n-1}[i]$. We easily check that $0 \leq k \leq 5$. In order to compute a carry-save number $R[i]$ congruent to $2R[i+1]+a_iB$ modulo F, it suffices to store the six possible values of $U[i] = (k \cdot 2^n) \bmod F$ in a table and to add this unsigned integer to $T^{(s)}_{n-1:0}[i] + 2T^{(c)}_{n-2:0}[i]$ by means of a second CSA. Note that the least significant bit of $T^{(c)}[i]$ is always equal to zero and that $R[i] \leq 2^{n+1} + F - 5$. This operator seems more attractive than the one by Jeong and Burleson [18]: at the price of a slightly more complex table, the iteration stage requires two CSAs instead of three. The final modular reduction remains unfortunately expensive and can be computed with the *Modred* block illustrated on Figure 7.

Amanor *et al.* showed that, if both B and F are constants known at design time, the iteration stage consists of a table and a single CSA [22] (Figure 11). Since the original algorithm requires an even more complex final modulo F reduction, we describe here a slightly modified version which allows one to perform this operation with the *Modred* block depicted by Figure 7. $R^{(s)}[i+1]$ and $R^{(c)}[i+1]$ are again two n-bit integers and the computation of a number $R[i]$ congruent to $2R[i+1] + a_iB$ is carried out according to Equation (1) with $\alpha = n-1$. The table is addressed by a_i and the 3-bit sum $k = R^{(c)}_{n-1:n-2}[i+1] + r^{(s)}_{n-2}[i+1]$, and returns an n-bit integer $(a_iB + k2^n) \bmod F$.

B. *Radix*-2 *Algorithms*

Beuchat and Muller proposed two non-redundant radix-2 versions of Kim and Sobelman's recurrence in [23]. These algorithms are designed for the modular multiplication of operands up to 32 bits on FPGAs embedding dedicated carry logic. The first scheme carries out $(AB+C) \bmod F$ according to the following iteration:

$$\begin{aligned} S[i] &= 2R[i+1], \\ T[i] &= S[i] + c_i + a_iB, \\ R[i] &= \varphi(T[i] \text{ div } 2^n) + T[i] \bmod 2^n, \end{aligned} \tag{5}$$

where $\varphi(k) = (2^n \cdot k) \bmod F$ (Figure 12). The main problem consists in finding the maximal values of $R[i]$ and $T[i]$, on which depends the size of the table implementing the $\varphi(k)$ function. Contrary to algorithms in redundant number systems, for which one can only compute a rough estimate by now, the nonlinear recurrence relation defined by Equation (5) has been solved. This result allows one to establish several nice properties of the algorithm. Assume that $A \in \mathbb{N}$, $B \in \{0, \ldots, F-1\}$, and $C \in \mathbb{N}$. Then $T[i]$ is an $(n+2)$-bit number, $\forall F \in \{2^{n-1}+1, \ldots, 2^n - 1\}$, and the table is addressed by only two bits [23]. Furthermore, $\varphi(k)$ is defined recursively

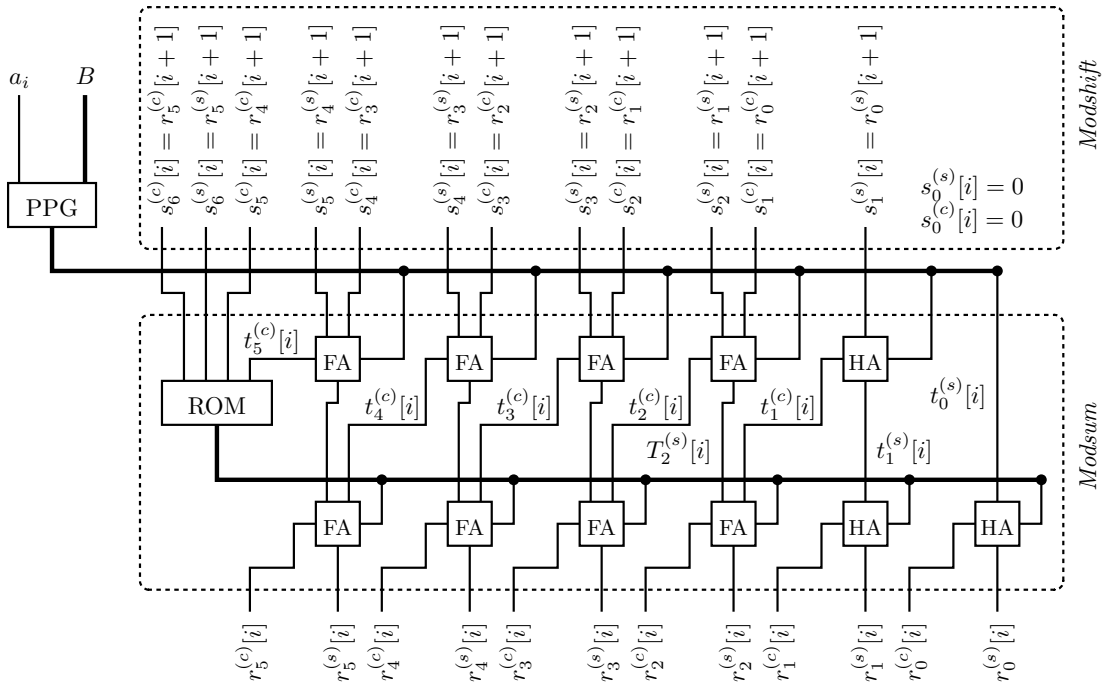


Fig. 10. Architecture of the iteration stage proposed by Kim and Sobelman [21] for $n = 6$.

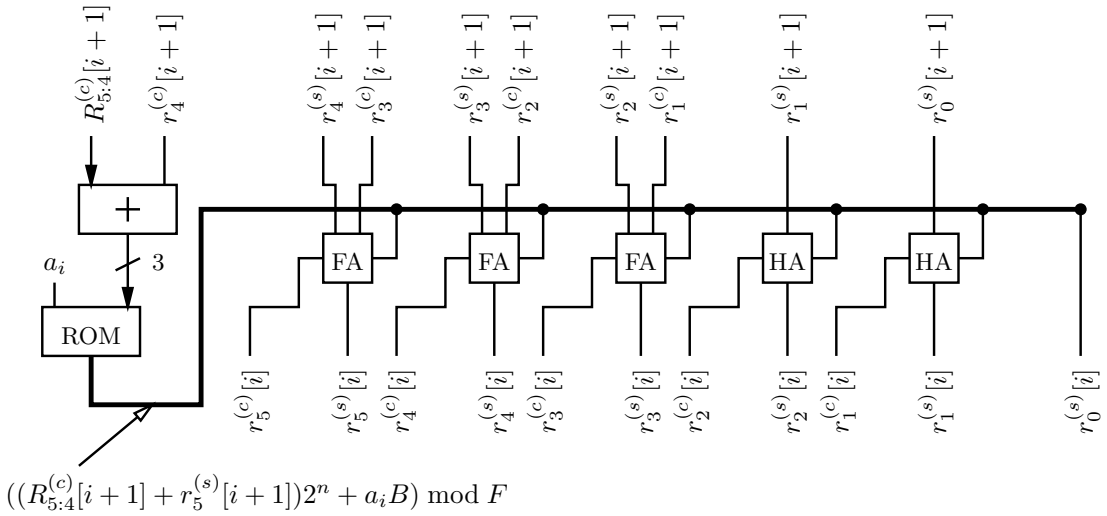


Fig. 11. Architecture of the iteration stage proposed by Amanor *et al.* [22] for $n = 6$.

on $\mathbb{N}^*$ as follows:

$$\varphi(k) = \begin{cases} \varphi(k-1) - 2F + 2^n & \text{if } \varphi(k-1) - 2F + 2^n \geq 0, \\ \varphi(k-1) - F + 2^n & \text{otherwise,} \end{cases} \tag{6}$$

with $\varphi(0) = 0$. Note that two CRAs, an array of n AND gates, and three registers implement the above equation (Figure 12). Thus, the critical path is the same as the one of the circuit implementing the iteration stage. Note that, at the price of an additional clock cycle, one can build the table on-the-fly without impacting on the computation time (Figure 12). The algorithm returns a number $R[0]$ congruent to $(AB + C)$ modulo F and a final modular reduction is required. The architecture of the circuit responsible for this operation depends on F: if $2^{n-1} + 1 \leq F \leq 2^{n-1} + 2^{n-2} - 1$, one shows that $R[0] < 3F$; if $2^{n-1} + 2^{n-2} \leq F \leq 2^n - 1$, then $R[0] < 2F$ (Figure 12).

And yet, this first radix-2 algorithm has a drawback in the sense that $R[0]$ is not a valid input. Since both right-to-left and left-to-right modular exponentiation algorithms involve the computation of $(R[0]^2) \bmod F$, a modulo F reduction is required at the end of each multiplication. A straightforward modification of the algorithm solves this issue: it suffices to compute $R[i] = \psi(T[i] \operatorname{div} 2^{n-1}) + T[i] \bmod 2^{n-1}$, where $\psi(k) = (2^{n-1} \cdot k) \bmod F$. Let $B_{\max} = \frac{2^{n+2}+11-4\cdot(n \bmod 2)}{3}$. Assume that $A \in \mathbb{N}$, $B \in \{0, \ldots, Y_{\max}\}$, and $C \in \mathbb{N}$. Then, one can establish the following properties [23]:

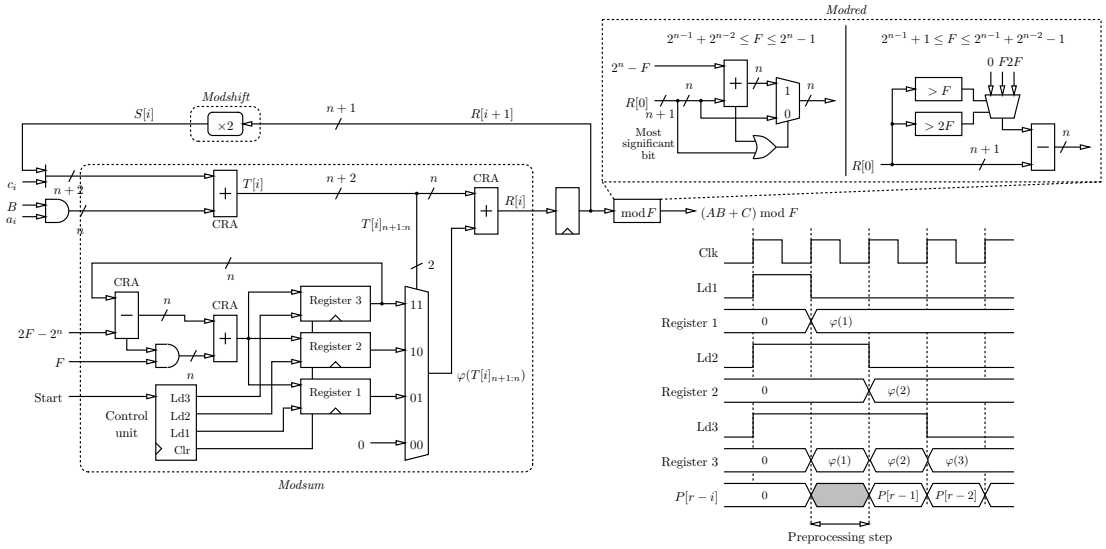


Fig. 12. Architecture of the first iteration stage proposed by Beuchat and Muller [23].

- $T[i]$ is an $(n+2)$-bit number, $\forall F \in \{2^{n-1}+1, \ldots, 2^n - 1\}$, and the ψ table is addressed by three bits. Furthermore, one can also build the table on-the-fly at the price of an extra clock cycle (Figure 13).
- $R[0]$ is smaller than $2F$ and at most one subtraction is required to compute $AB \bmod F$ from $R[0]$.
- $R[0]$ is smaller than $B_{\max}$. Therefore modular exponentiation can be performed with $R[0]$ instead of R.

Further optimizations are possible when the modulus F is known at design time. Figure 14a describes the implementation of the φ function on Xilinx FPGAs. In this example, the operator is able to perform multiplication-addition modulo F_1 or F_2 according to a *Select* signal. Thus, each bit of φ is computed by means of a 3-input table addressed by $T_{n+1}[i]$, $T_n[i]$, and *Select*. Such tables are embedded in the LUTs of the CRA returning $R[i]$. The ψ function is implemented the same way (Figure 14b). However, since it depends on three bits, the operator handles a single modulus F.

VI. Third Architecture: Modular Reduction Followed by a Left-Shift Operation

The third family of algorithms aims at simplifying the final modular reduction at the price of an additional iteration. This elegant approach was introduced by Peeters *et al.* [20] and can be applied to both prime fields and extension fields. Let us consider multiplication over $\mathbb{F}_p$ to illustrate how such architectures work out the product $AB \bmod F$. The *Modshift* block computes a number $U[i]$ congruent to $R[i+1]$ modulo F and returns an even number $S[i] = 2U[i]$. Recall that algorithms based on Horner's rule compute a number $R[0] = AB + \alpha F$ congruent to AB modulo F, where $\alpha \in \mathbb{N}$. Let us perform an additional iteration with $a_{-1} = 0$. We have $R[-1] = S[-1] = 2(R[0] - \beta F) = 2(AB + (\alpha - \beta)F)$. Since $R[-1]$ is even, we can shift it to get $R[-1]/2 = AB + (\alpha - \beta)F$, which is congruent to AB modulo F. Furthermore, the upper bound of $R[-1]$ turns out to be smaller than the one of $R[0]$.

A. Carry-Save Algorithm

The first carry-save modular multiplier featuring such a *Modshift* block was probably proposed by Bunimov and Schimmler [19]. However, this algorithm requires an $(n+2)$-bit integer $R^{(s)}[i]$ and an $(n+1)$-bit integer $R^{(c)}[i]$ to encode the intermediate result $R[i]$. Since the authors do not perform an additional iteration, the final modular reduction proves to be more complex than the one of the carry-save modular multipliers studied in Section V. Peeters *et al.* designed a carry-save architecture which returns either $R[-1] = AB \bmod F$ or $R[-1] = (AB \bmod F) + F$ [20]. The carry-save intermediate result $R[i]$ consists of an $(n+1)$-bit word $R^{(s)}[i]$ and an n-bit word $R^{(c)}[i]$, whose least significant bit is always equal to zero (i.e. $R^{(c)}[i] \leq 2^n - 2$). Therefore, we have

$$\begin{aligned} R[i] &= \left(R^{(s)}_{n:n-2}[i] + R^{(c)}_{n-1:n-3}[i]\right) \cdot 2^{n-2} + \\ &\quad R^{(s)}_{n-3:0}[i] + 2R^{(c)}_{n-4:0}[i]. \end{aligned}$$

Let us define the four bit integer $U[i]$ such that $U[i] = R^{(s)}_{n:n-2}[i] + R^{(c)}_{n-1:n-3}[i]$. The *Modshift* block computes an number

$$\begin{aligned} S[i] &= 2\left(U[i+1] \cdot 2^{n-2}\right) \bmod F + \\ &\quad 2R^{(s)}_{n-3:0}[i+1] + 4R^{(c)}_{n-4:0}[i+1], \end{aligned}$$

which is congruent to $R[i+1]$ modulo F. However, Peeters *et al.* do not compute $S[i]$ explicitly. They suggest to evaluate $(k \cdot 2^{n-2}) \bmod F$ and $2R^{(s)}_{n-3:0}[i+1] + 4R^{(c)}_{n-4:0}[i+1] + a_i B$ in parallel in order to shorten the critical path (Figure 15). The modulus must be known at design time in order to build the table storing the 15 possible values of $(U[i+1] \cdot 2^{n-2}) \bmod F$. Note that $a_i B \leq F - 1$, $(U[i+1] \cdot 2^{n-2}) \bmod F \leq F - 1$, $R^{(s)}_{n-3:0}[i+1] \leq 2^{n-2} - 1$, and $R^{(c)}_{n-4:0} \leq 2^{n-3} - 2$. The number $R[0]$ is therefore smaller than or equal to $3F + 2^n - 13$ and the *Modred* block would have to subtract up to $4F$ to get the final result. Let us perform an additional iteration with $a_{-1} = 0$. We obtain an even number $R[-1] \leq 2F + 2^n - 12$ which is

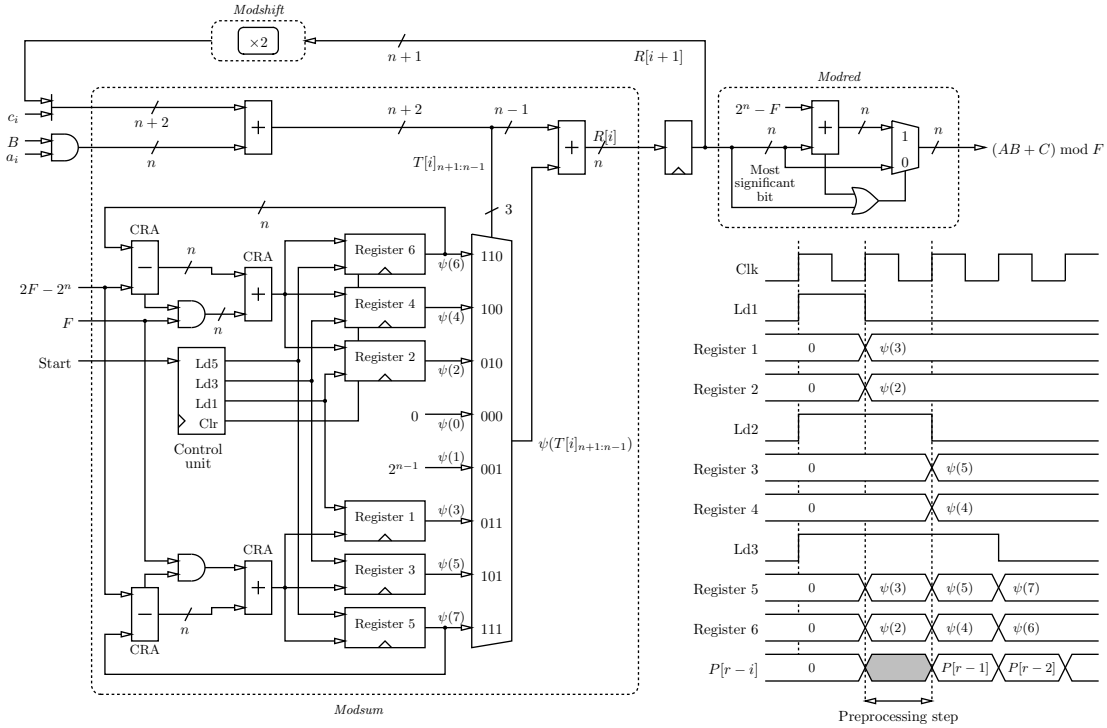


Fig. 13. Architecture of the second iteration stage proposed by Beuchat and Muller [23].

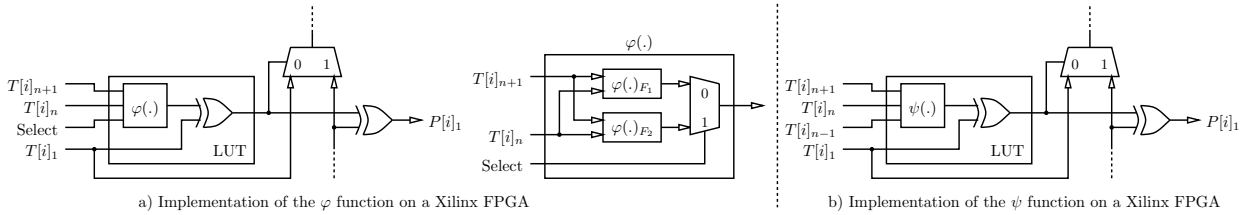


Fig. 14. Optimizations of the algorithm proposed by Beuchat and Muller [23] when the modulus is known at design time.

congruent to $2AB$ modulo F. Therefore, we have to reduce $R[-1]/2 \leq F + 2^{n-1} - 6 < 2M$. This operation requires at most one subtraction.

B. High-Radix Carry-Save Algorithm

Since carry-save addition does not take advantage of the dedicated carry-logic available in several FPGA families, Beuchat and Muller [11] proposed a high-radix carry-save implementation of the algorithm by Peeters *et al.* [20] previously described. Assume that $R[i+1]$ and $S[i]$ are now high-radix carry-save numbers. By shifting $R[i+1]$, we define a new internal representation for $S[i]$. It is therefore necessary to perform a conversion while computing a number $R[i]$ congruent to $S[i] + a_i B$ modulo F. Beuchat and Muller showed that the amount of hardware required for this task depends on the encoding of $R[i]$ and the modulus F [11]. They also proposed an algorithm which selects the optimal high-radix carry-save number system and generates the VHDL description of the modular multiplier. Such operators perform a multiplication in $(n+1)$ clock cycles and return a high-radix carry save number $R[-1]$ which is smaller than $2F$. Thus, the final modulo F correction requires at most one subtraction.

C. Multiplication over $\mathbb{F}_{p^n}$

The same approach allows one to design array multipliers over $\mathbb{F}_{p^n}$. Song and Parhi suggested to compute at each step a degree-$(n + D - 2)$ polynomial $T[i]$ which is the sum of D partial products, i.e. $T[i] = \sum_{j=0}^{D-1} a_{Di+j} x^j B$ [1] (Figure 8b). A degree-$(n+D-1)$ polynomial $S[i]$ allows one to accumulate these partial products: $S[i] = T[i] + x^D (S[i+1] \bmod F)$. After $\lceil n/D \rceil$ iterations, $S[0]$ is a degree-$(n + D - 1)$ polynomial congruent to AB modulo F. Song and Parhi included specific hardware to carry out a final modular correction. However, we achieve the same result by performing an additional iteration with $a_{-1} = 0$ [25]. Since $T[-1]$ is equal to zero, we obtain $R[-1] = S[-1] = x^D (AB \bmod F)$ and it suffices to right-shift this polynomial to get the result.

VII. Conclusion

In order to compare the algorithms described in this survey, we wrote a generic VHDL library as well as automatic code generators, and performed a series of experiments involving

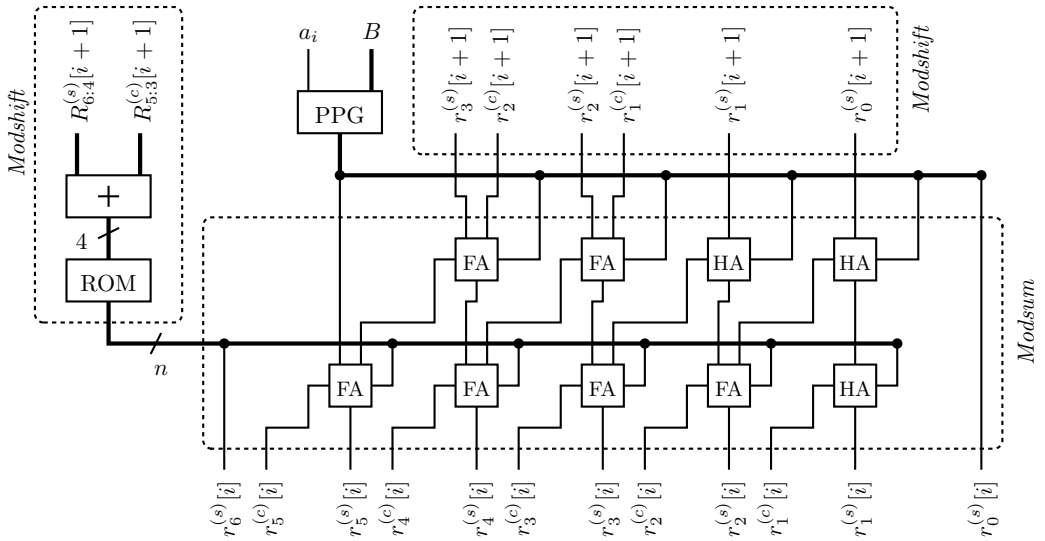


Fig. 15. Architecture of the iteration stage proposed by Peeters *et al.* [20] for $n = 6$.

a Spartan-3 XC3S1500 FPGA. Whereas the description of operators whose modulus is an input is rather straightforward, the computation of the tables involved in the algorithms for which the modulus is a constant known at design time proves to be tricky in VHDL. Since the language does not allow one to easily deal with big numbers, a first solution consists in writing a VHDL package for arbitrary precision arithmetic. Note that this approach slows down the synthesis of the VHDL code. Consider for instance the computation of the $\varphi(k)$ function involved in the radix-2 algorithm (see Equation (6) in Section V-B). Synthesis tools have to interpret the code of the recursive function $\varphi(k)$ in order to compute the constants $(k \cdot 2^n) \bmod F$. In some cases, it seems more advisable to write a program which automatically generates the VHDL description of the operator according to its modulus: the selection of a high-radix carry-save number system for the algorithm outlined in Section VI-B consists for instance in finding a shortest path in a directed acyclic graph [11].

Figure 16 describes a comparison between carry-save and radix-2 iteration stages when the modulus is a constant. Among carry-save algorithms, the one by Kim and Sobelman [21] leads to the smallest iteration stage. However, recall that it involves a complex *Modred* block and the architecture introduced by Peeters *et al.* [20] proves to be the best choice. The operator introduced by Jeong and Burleson [18] requires a larger area and is even slower than other carry-save implementations. Radix-2 algorithms take advantage of the dedicated carry logic and embed the $\varphi(k)$ table in the LUTs of a CRA (Figure 13). This approach allows one to roughly divide by two the area on Xilinx devices at the price of a slightly lower clock frequency. Since these results do not include the *Modred* block, the delay of carry-save operators is underestimated. However, these results indicate that radix-2 algorithms are efficient for moduli up to 32 bits. For larger moduli, the high-radix carry-save approach allows significant hardware savings without impacting on the computation time on Xilinx FPGAs (Figure 17). Note that borrow-save algorithms always lead to larger circuits on our target FPGA family. Experiment results indicate that the choice between the multipliers over GF(p^m) studied in this paper depends on the irreducible polynomial F (see also [26]).

ACKNOWLEDGMENTS

The authors would like to thank Nicolas Brisebarre and Jérémie Detrey for their useful 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).

REFERENCES

[1] L. Song and K. K. Parhi, "Low energy digit-serial/parallel finite field multipliers," *Journal of VLSI Signal Processing*, vol. 19, no. 2, pp. 149–166, July 1998.

[2] S. E. Erdem, T. Yamk, and Ç. K. Koç, "Polynomial basis multiplication over GF(2^m)," *Acta Applicandae Mathematicae*, vol. 93, no. 1–3, pp. 33–55, Sept. 2006.

[3] J. Guajardo, T. Güneysu, S. Kumar, C. Paar, and J. Pelzl, "Efficient hardware implementation of finite fields with applications to cryptography," *Acta Applicandae Mathematicae*, vol. 93, no. 1–3, pp. 75–118, Sept. 2006.

[4] S. Kumar, T. Wollinger, and C. Paar, "Optimum digit serial GF(2^m) multipliers for curve-based cryptography," *IEEE Transactions on Computers*, vol. 55, no. 10, pp. 1306–1311, Oct. 2006.

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

[6] M. E. Kaihara and N. Takagi, "Bipartite modular multiplication," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, ser. Lecture Notes in Computer Science, J. R. Rao and B. Sunar, Eds., no. 3659. Springer, 2005, pp. 201–210.

[7] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "A RNS Montgomery modular multiplication algorithm," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 766–776, July 1998.

[8] Ç. K. Koç and C. Y. Hung, "Multi-operand modulo addition using carry save adders," *Electronics Letters*, vol. 26, no. 6, pp. 361–363, Mar. 1990.

[9] ——, "Carry-save adders for computing the product AB modulo N," *Electronics Letters*, vol. 26, no. 13, pp. 899–900, June 1990.

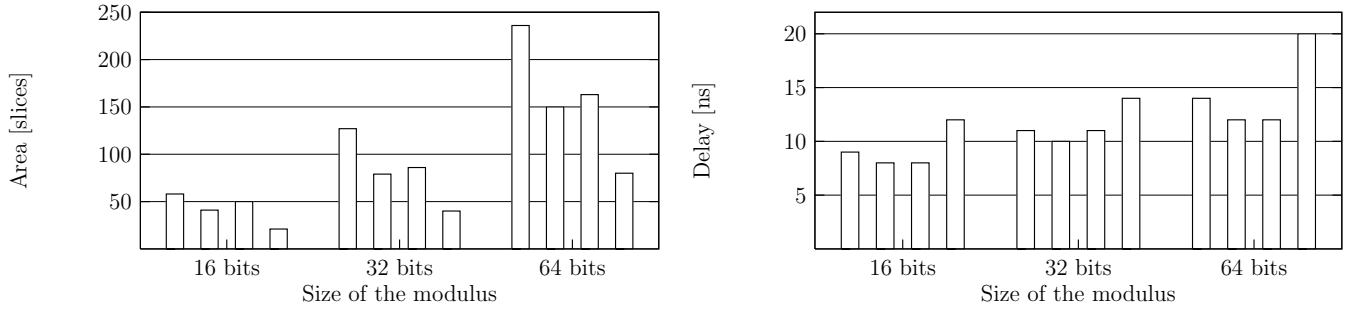


Fig. 16. Comparison between carry-save and radix-2 algorithms for several operand sizes. For each experiment, we consider, from left to right, the algorithms by Jeong and Burleson [18] (carry-save), Kim and Sobelman [21] (carry-save), Peeters *et al.* [20] (carry-save), and Beuchat and Muller [23] (radix-2).

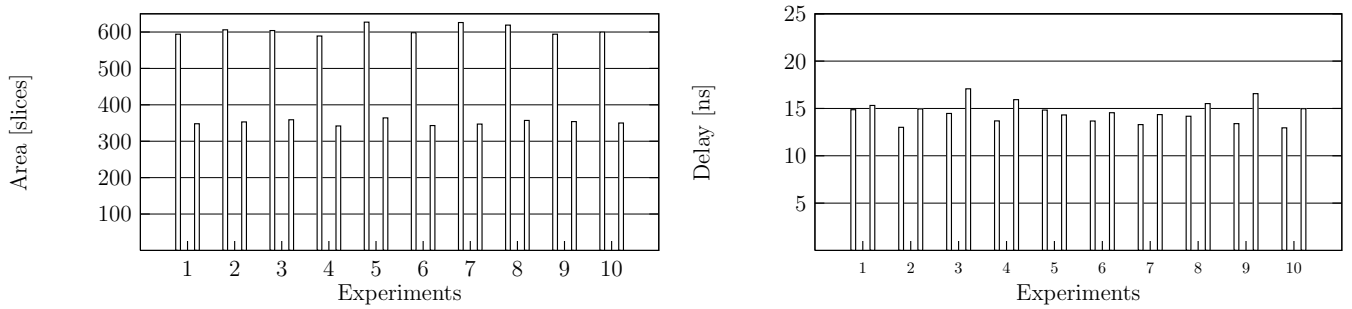


Fig. 17. Comparison between the carry-save algorithm proposed by Peeters *et al.* [20] and the high-radix carry-save scheme by Beuchat and Muller [11]. Ten 256-bit prime moduli were randomly generated for this experiment.

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

[11] J.-L. Beuchat and J.-M. Muller, "Automatic generation of modular multipliers for FPGA applications," Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, Tech. Rep. 2007–1, Jan. 2007.

[12] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. 10, 1961.

[13] J.-C. Bajard, J. Duprat, S. Kla, and J.-M. Muller, "Some operators for on-line radix-2 computations," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 336–345, 1994.

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

[15] P. Grabher and D. Page, "Hardware acceleration of the Tate Pairing in characteristic three," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, ser. Lecture Notes in Computer Science, J. R. Rao and B. Sunar, Eds., no. 3659. Springer, 2005, pp. 398–411.

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

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

[18] Y.-J. Jeong and W. P. Burleson, "VLSI array algorithms and architectures for RSA modular multiplication," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 5, no. 2, pp. 211–217, June 1997.

[19] V. Bunimov and M. Schimmler, "Area and time efficient modular multiplication of large integers," in *Proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, Eds. IEEE Computer Society, 2003, pp. 400–409.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] C. Shu, S. Kwon, and K. Gaj, "FPGA accelerated Tate pairing based cryptosystem over binary fields," 2006, cryptology ePrint Archive, Report 2006/179.

[25] J.-L. Beuchat, M. Shirase, T. Takagi, and E. Okamoto, "An algorithm for the η_T pairing calculation in characteristic three and its hardware implementation," 2006, cryptology ePrint Archive, Report 2006/327.

[26] J.-L. Beuchat, T. Miyoshi, Y. Oyama, and E. Okamoto, "Multiplication over $\mathbb{F}_{p^m}$ on FPGA: A survey," in *Reconfigurable Computing: Architectures, Tools and Applications – Proceedings of ARC 2007*, ser. Lecture Notes in Computer Science, P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. M. P. Cardoso, Eds., no. 4419. Springer, 2007, pp. 214–225.