A Bit-Serial Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m)

Johann Großschädl

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
Johann.Groszschaedl@iaik.at

Abstract. The performance of elliptic curve cryptosystems is primarily determined by an efficient implementation of the arithmetic operations in the underlying finite field. This paper presents a hardware architecture for a unified multiplier which operates in two types of finite fields: GF(p) and GF(2^m). In both cases, the multiplication of field elements is performed by accumulation of partial-products to an intermediate result according to an MSB-first shift-and-add method. The reduction modulo the prime p (or the irreducible polynomial $p(t)$, respectively) is interleaved with the addition steps by repeated subtractions of $2p$ and/or p (or $p(t)$, respectively). A bit-serial multiplier executes a multiplication in GF(p) in approximately $1.5 \cdot \lceil \log_2(p) \rceil$ clock cycles, and the multiplication in GF(2^m) takes exactly m clock cycles. The unified multiplier requires only slightly more area than that of the multiplier for prime fields GF(p). Moreover, it is shown that the proposed architecture is highly regular and simple to design.

Keywords: Elliptic curve cryptography, finite field arithmetic, iterative modulo multiplication, polynomial basis representation, bit-serial multiplier architecture, smart card crypto-coprocessor.

1 Introduction

In the mid-eighties, N. Koblitz [9] and V. S. Miller [16] independently proposed using the group of points on an elliptic curve (EC) over a finite field in discrete logarithm cryptosystems. Elliptic curve cryptography can be used to provide digital signature schemes, encryption schemes, and key agreement schemes [10]. The primary advantage of elliptic curve systems over systems based on the multiplicative group of a finite field is the absence of a subexponential-time algorithm that could solve the discrete logarithm problem (DLP) in these groups [3]. Consequently, an elliptic curve group that is smaller in size can be used, while maintaining the same level of security [13]. The result is smaller key sizes, bandwidth savings, and faster implementations. These features make elliptic curve cryptosystems especially attractive for applications in environments where computational power is limited, such as smart cards or hand-held devices.

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 202–219, 2001.
© Springer-Verlag Berlin Heidelberg 2001

The performance of an elliptic curve cryptosystem is primarily determined by the efficient realization of the arithmetic operations (addition, multiplication, and inversion) in the underlying finite field. Many practical implementations use *projective coordinates* [15] to represent points on the elliptic curve because they allow to perform a point addition/doubling without inversion. Therefore, coprocessors for elliptic curve cryptography are most frequently designed to accelerate the field multiplication.

1.1 Motivation for a Unified Multiplier Architecture

An elliptic curve can be defined over various mathematical structures such as a ring or field. In cryptography only finite fields are used because they allow to store and handle the field elements in a manageable way. Due to standardization activities, two special types of finite fields have become very important for the implementation of elliptic curve cryptosystems: The *prime field* GF(p) and the *binary extension field* GF(2^m). Various accredited standards bodies like the *National Institute of Standards and Technology* (NIST) recommended to use either GF(p) or GF(2^m) as the underlying finite field [19]. In order to promote interoperability between different implementations and to facilitate widespread use of well-accepted techniques, a crypto-coprocessor should operate in both types of finite fields. Therefore, it is an obvious idea to develop a unified multiplier architecture which can perform multiplications in GF(p) and GF(2^m). At a first glance, prime fields and binary extension fields seem to have dissimilar properties. However, the elements of either field can be represented using a bit-string. Furthermore, the arithmetic operations in both fields have structural similarities allowing a unified design. For example, a multiplication in GF(p) is performed modulo a prime p, and the multiplication in GF(2^m) is done modulo an irreducible polynomial $p(t)$ if polynomial basis representation is used.

1.2 Previous Work

In August 2000, E. Savaş *et al.* introduced a unified multiplier which operates in both types of finite fields, GF(p) and GF(2^m) [23]. From an algorithmic point of view, the multiplication in GF(p) is performed according to Montgomery's method [17]. The introduction of the Montgomery multiplication for the field GF(2^m) in [11] opened them up the possibility to develop a unified multiplier architecture by taking advantage of the fact that the Montgomery multiplication is in both fields essentially the same operation. Their implementation utilizes inherent concurrency in Montgomery multiplication and uses an array of word-size processing units organized in a pipeline. Savaş' architecture is highly scalable because a fixed-area multiplier can handle operands of any size. Moreover, the word-size of a processing unit as well as the number of pipeline stages can be selected according to the desired area/performance trade-off.

Another interesting VLSI implementation was reported by J. Goodman *et al.* [6]. Their so-called Domain Specific Reconfigurable Cryptographic Processor (DSRCP) provides a full suite of arithmetic operations (including inversion) over the integers modulo p, binary extension fields, and non-supersingular elliptic curves over GF(2^m), with operands ranging in size from 8 to 1024 bits. These operations are implemented using a single computation unit whose datapath cells can be reconfigured on the fly. The modulo multiplication is realized according to an iterated radix-2 version of Montgomery multiplication. On the other hand, the multiplication in GF(2^m) is based on an iterated MSB-first approach.

1.3 Our Contribution

We introduce a multiplier architecture for unified (dual-field) arithmetic. The modulo multiplication proceeds in a serial-parallel fashion according to an iterative approach, which means that the modulo reduction is performed during multiplication through concurrent reduction of the intermediate result.

The main contribution of this paper is a modification of the classical MSB-first version for iterative modulo multiplication that allows a very efficient hardware implementation. Additionally, we propose a bit-serial architecture using carry-save adders for the accumulation of partial-products to an intermediate result given in a redundant representation. The modulo reduction operation is interleaved with the partial-product additions by repeated subtractions of once or twice the modulus. The circuit to decide the multiple of the modulus to be subtracted is very simple and requires only the two highest order bits of the redundant intermediate result as inputs. Contrary to other designs, the subtrahend evaluation circuit of our multiplier does not cause a significant critical path.

We will show that the bit-serial multiplier can also perform multiplications in GF(2^m) by simply setting all carry-bits of the intermediate result to 0. The area-cost of the unified multiplier is only slightly higher than that of the multiplier for the field GF(p), providing significant area savings when both types of multiplier are needed. To the best of our knowledge, an MSB-first bit-serial architecture for multiplication in GF(p) and GF(2^m) has never been published before. Compared to the Montgomery multiplication used in Savaş' implementation, the MSB-first iterative algorithm requires neither a transformation of operands into Montgomery domain nor precomputed constants. The bit-serial architecture has a linear array structure with a bit-slice feature. A high degree of regularity and mainly local connections make the multiplier simple to design.

1.4 Paper Outline

The remainder of this paper is organized as follows: Section 2 provides some background information on MSB-first techniques for radix-2 multiplication with interleaved reduction. Section 3 presents a modified version of the classical "shift-and-add" algorithm for modulo multiplication. The modified algorithm uses a redundant representation of the intermediate result and profits from a novel quotient estimation technique which is detailed in subsection 3.1. Section 4 covers arithmetic in binary extension fields GF(2^m) using a polynomial basis representation. The unified multiplier architecture for GF(p) and GF(2^m) is introduced in section 5. This section also describes the execution of a multiplication and presents an estimation of the computation time for both types of finite fields. The paper finishes with a summary of results and conclusions in section 6.

2 Preliminaries

The finite field GF(p), also denoted as *prime field* of order p, is the field of residue classes modulo p, where the field elements are the integers $0, 1, \ldots, p-1$. The field operations are modulo operations, i.e., addition and multiplication modulo the prime p. Beside the popular Montgomery multiplication [17] and the Barret modulo reduction method [1], also binary and higher-radix algorithms for MSB-first iterative modulo multiplication have been proposed.

2.1 MSB-First Iterative Modulo Multiplication

A usual way of multiplying two integers A and B is done by scanning the multiplier B one bit at a time, beginning with the most significant bit (MSB), and accumulating the partial-product $A \cdot B[i]$ to the intermediate result. The product P is a $2n$-bit integer if the operands are n bits long and can be written as

$$P = A \cdot B = A \cdot \left(\sum_{i=0}^{n-1} B[i]\, 2^i \right) = \sum_{i=0}^{n-1} (A \cdot B[i])\, 2^i \tag{1}$$

The notation $X[i]$ indicates the i-th bit of an n-bit integer X; $X[0]$ is the LSB, and $X[n-1]$ is the MSB. After each addition of a partial-product, the intermediate result must be multiplied by 2 to align it to the next partial-product. Since a multiplication by 2 is a 1-bit left-shift in hardware, the described method is also known as *shift-and-add* multiplication. The shift-and-add multiplication typically results in a bit-serial architecture when implemented in hardware. Bit-serial multipliers offer a fair area/performance trade-off, which is an important aspect in the design of coprocessors for area-restricted devices like smart cards.

INPUT: An n-bit modulus M (i.e., $2^{n-1} \leq M < 2^n$), a multiplicand $A < M$, and a multiplier $B < M$.
OUTPUT: Result $R = A \cdot B \bmod M$.

$R \leftarrow 0$
for i **from** $n-1$ **downto** 0 **do**
    $R \leftarrow 2 \cdot R + A \cdot B[i]$
    $q \leftarrow \lfloor R/M \rfloor$
    $R \leftarrow R - q \cdot M$
endfor

Fig. 1: MSB-first shift-and-add multiplication with interleaved modulo reduction.

Figure 1 shows that the simple shift-and-add multiplication can be easily extended to perform a modulo multiplication. The modulo reduction of the intermediate result R is interleaved with the addition steps and realized by subtraction of the product $q \cdot M$, whereby q is the quotient of R and the modulus M. The quotient q can be at most 2 since the term $2 \cdot R + A \cdot B[i]$ is always smaller than three times the modulus M (on condition that $A < M$):

$$q = \left\lfloor \frac{R}{M} \right\rfloor \quad \text{with} \quad q \in \{0, 1, 2\} \tag{2}$$

Therefore, the reduction of the intermediate result can be accomplished by subtraction of M or $2 \cdot M$ (i.e., addition of the two's complement of M or $2 \cdot M$). However, two serious problems arise when implementing this algorithm:

1. Addition of long integers can cause a significant delay due to carry propagation from LSB to MSB, which limits the clock frequency.
2. The exact comparison of the intermediate result R to the modulus M in order to decide whether the quotient q is 0, 1 or 2 is also difficult to perform for very long integers.

Various papers on the efficient implementation of MSB-first modulo multiplication can be found in literature. An algorithm published by G. R. Blakley realizes the reduction of the intermediate result by one or two subtractions of the modulus [4]. E. F. Brickell presented an architecture which performs a multiplication of two integers modulo p in $\lceil \log_2(p) \rceil + 7$ clock cycles [5]. This approach uses delayed carry adders to avoid the carry propagation delay, but has problems due to the difficulty of comparing long integers and conversion of the result from delayed carry representation to binary representation. C. D. Walter proposed another technique for speeding up modulo multiplication by scaling the modulus [26]. The modulus is scaled in such a way that a certain number of the most significant digits are fixed, resulting in a simplified reduction operation. However, the cost of this method is precalculation and storage of the scaled modulus. Y.-J. Jeong *et al.* presented an architecture for iterative modulo multiplication that performs the quotient estimation by table lookups [8]. Their design also requires storage of some precalculated complements of the modulus, resulting in an increase in needed resources. The partial-parallel multiplier introduced by H. Orup *et al.* contains a quotient estimation circuit that estimates the 12 highest order bits of the redundant partial sum, and then chooses an appropriate multiple of the modulus to be subtracted [20]. The most significant drawback of Orup's architecture is a long critical path introduced by the quotient estimation circuit, which limits the clock frequency. Higher-radix methods for MSB-first iterative modulo multiplication have been reported in [12,18,24,25].

2.2 Carry-Save Adders

The carry propagation in long integer addition is easily eliminated by the implementation of a *carry-save adder* (CSA). Carry-save adders are widely used in arithmetic circuits due to their performance in terms of speed and silicon area [21]. An n-bit CSA consists of n full-adders (FA), and solves the carry propagation problem by using a *redundant representation* for the result (i.e., the carries are saved). This means that the result is not a single binary number, but is represented by two n-bit numbers instead: R_S (the sum bits) and R_C (the carry bits). The delay of a carry-save adder is constant (i.e., independent of the length of the operands) and only determined by the delay of a single full-adder. In many applications, the sum output R_S and the carry output R_C are latched or registered, either for synchronization purposes or for pipelining.

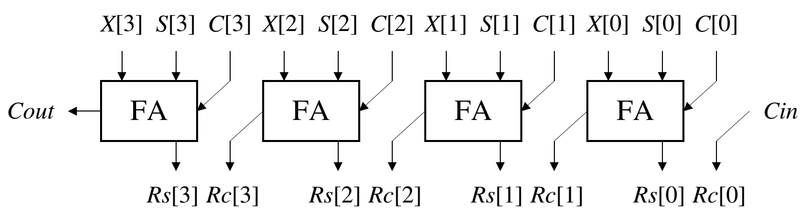

Fig. 2: Block diagram of a 4-bit carry-save adder.

Figure 2 illustrates a 4-bit carry-save adder. The basic principle of the carry-save addition is to reduce the sum of three binary numbers S, C, X to the sum of two binary numbers R_S, R_C without carry propagation according to the following equations:

$$R_S[i] = S[i] \otimes C[i] \otimes X[i] \tag{3}$$

$$R_C[i+1] = S[i] \cdot C[i] + S[i] \cdot X[i] + C[i] \cdot X[i] \quad \text{with} \quad R_C[0] = Cin = 0 \tag{4}$$

Note that the operators in the previous equations are logical operators and not arithmetic operators. When using carry-save adders, the intermediate result R is not a single binary number anymore, but is given in a redundant representation as a *sum and carry pair* (R_S, R_C) instead, whereby R_S denotes the sum part of the result, and R_C the carry part, respectively. Carry-save adders are advantageous if many subsequent additions have to be performed.

3 Optimized MSB-First Iterative Modulo Multiplication

The major hindrance of the bit-serial architectures for modulo multiplication described in subsection 2.1 is that they either require a costly quotient evaluation circuit or a circuit for performing comparisons of long integers. These circuits cause significant additional hardware and may limit the clock frequency due to a long critical path. Furthermore, some of the mentioned implementations need a large amount of storage for precomputed multiples of the modulus. If the modulus is to be dynamic, the stored modulus multiples must be updated whenever the modulus is changed.

INPUT: An n-bit modulus M (i.e., $2^{n-1} \leq M < 2^n$), a multiplicand A in the range of $0 \leq A < 2^n$, and a multiplier B in the range of $0 \leq B < 2^n$.
OUTPUT: The result R in the range of $0 \leq R < 2^n$. R is possibly not fully reduced, i.e., $R = A \cdot B \bmod M + k \cdot M$ with $k \in \{0, 1\}$.

$(R_S, R_C) \leftarrow 0$
for i **from** $n-1$ **downto** 0 **do**
  $(R_S, R_C) \leftarrow 2 \cdot (R_S, R_C) + A \cdot B[i]$
  while $(R_S, R_C) \geq 2 \cdot 2^n$ **do** $(R_S, R_C) \leftarrow (R_S, R_C) - 2 \cdot M$
  while $(R_S, R_C) \geq 2^n$ **do** $(R_S, R_C) \leftarrow (R_S, R_C) - M$
endfor
$R \leftarrow R_S + R_C$ { red. to non-red. conversion }
if $R \geq 2^n$ **then**
  $(R_S, R_C) \leftarrow R - M$
  $R \leftarrow R_S + R_C$ { red. to non-red. conversion }
endif

Fig. 3: Optimized version of the MSB-first iterative modulo multiplication.

The most crucial operation of the classical MSB-first algorithm for iterative modulo multiplication is the calculation of the quotient q, which is the same as to decide whether the current intermediate result is smaller than M (and consequently $q = 0$), or bigger than M (and consequently $q = 1$), or bigger than $2 \cdot M$ (and consequently $q = 2$). This decision is difficult for long integers because an n-bit modulus M can vary between its minimum value M_{min} of 2^{n-1} and its maximum value M_{max} of $2^n - 1$:

$$2^{n-1} \leq M < 2^n \;\Rightarrow\; M_{min} = 2^{n-1} \text{ and } M_{max} = 2^n - 1 \tag{5}$$

Additionally, a redundant representation of the intermediate result does not make this task easier. An efficient solution for this problem is to compare the redundant intermediate result to 2^n and to $2 \cdot 2^n$ instead of the exact values of M and $2 \cdot M$, since these comparisons are simpler to implement in hardware, as will be demonstrated in subsection 3.1.

Figure 3 shows a modified version of the shift-and-add multiplication which is optimized for hardware implementation. The intermediate result is written in redundant representation (R_S, R_C) to indicate that the additions and subtractions should be performed by carry-save adders. Another interesting detail of the modified algorithm is the fact that the modulo reduction is not carried out "at once", but is split into continued subtractions of $2 \cdot M$ and/or M. The subtraction of M and $2 \cdot M$ can be realized by addition of the two's complement of M, and by addition of the 1-bit left-shifted two's complement of M, respectively. When using a carry-save adder for the two's complement addition, the subtraction is performed in constant time. During a modulo multiplication the intermediate result is always in redundant representation. After the last multiplier bit $B[0]$ has been processed, the result must be converted from redundant into non-redundant representation. This conversion can be performed by a pipelined *carry-lookahead*

Table 1: Redundant number estimations. R_S and R_C are both (n+1)-bit numbers, $R_S[n]$ is the MSB of R_S, and $R_C[n]$ is the MSB of R_C.

$R_S[n]$	$R_C[n]$	$R_S[n{-}1]$	$R_C[n{-}1]$	$R_S + R_C$	Estimation
0	0	0	0	$R_S{+}R_C < 2^{n-1}{+}2^{n-1}$	$(R_S, R_C) < 3{\cdot}2^{n-1}$
0	0	0	1	$R_S{+}R_C < 2^{n-1}{+}2^{n}$	
0	0	1	0	$R_S{+}R_C < 2^{n}{+}2^{n-1}$	
0	0	1	1	$R_S{+}R_C \geq 2^{n-1}{+}2^{n-1}$	$(R_S, R_C) \geq 2^n$
0	1	X	X	$R_S{+}R_C \geq 2^n$	
1	0	X	X	$R_S{+}R_C \geq 2^n$	
1	1	X	X	$R_S{+}R_C \geq 2^n{+}2^n$	$(R_S, R_C) \geq 2{\cdot}2^n$

adder (see section 5). If the non-redundant result R is bigger than 2^n, one final subtraction of M is necessary to bound the result within the range of $[0, 2^n)$. It must be emphasized, however, that the operands A and B do not need to be fully reduced, but they must be smaller than 2^n to ensure that the algorithm works correctly. A remaining problem is the comparison of the redundant intermediate result to $2{\cdot}2^n$ and 2^n, respectively. In the next subsection we present an efficient solution for this problem by applying a special estimation technique.

3.1 Redundant Number Estimation

The modified algorithm in figure 3 requires a comparison of the intermediate result to $2{\cdot}2^n$ and 2^n to decide whether or not a subtraction of $2{\cdot}M$ or M has to be performed. For hardware implementation, this is a significant improvement over the first algorithm because it avoids the necessity for an exact comparison between the intermediate result and the modulus. Furthermore, the comparison to $2{\cdot}2^n$ and 2^n can be easily realized by a novel estimation technique, in the following denoted as *redundant number estimation.*

Table 1 shows a simplified logical truth-table to decide the two inequalities $(R_S, R_C) \geq 2^n$ and $(R_S, R_C) \geq 2{\cdot}2^n$. For decision of the first inequality, only the two most significant bits of R_S and R_C need to be scanned, and for the second inequality only the MSB of R_S and R_C, respectively. Note that R_S and R_C are both (n+1)-bit numbers, consequently the MSB of R_S is $R_S[n]$, and the MSB of R_C is $R_C[n]$. The hardware to decide the multiple of the modulus to be subtracted can be defined by the following two logical equations:

$$sub1 = R_S[n] + R_C[n] + (R_S[n-1] \cdot R_C[n-1]) \tag{6}$$

$$sub2 = R_S[n] \cdot R_C[n] \tag{7}$$

If $sub1 = 0$, then the intermediate result (R_S, R_C) is smaller than $3{\cdot}2^{n-1}$ and consequently also smaller than $3{\cdot}M$. This estimation is correct for any value of M according to equation (5), even for $M = M_{min}$. On the other hand, if $sub1 = 1$, the intermediate result is bigger than 2^n, and consequently it can be

estimated to be also bigger than M. Therefore, at least one subtraction of M is necessary, even if $M = M_{max}$.
For any n-bit modulus M satisfying $2^{n-1} \leq M < 2^n$, the redundant number estimations observed from table 1 can be summarized as follows:

$$sub1 = 0 \text{ and } sub2 = 0 \Rightarrow (R_S, R_C) < 3{\cdot}M \tag{8}$$

$$sub1 = 1 \text{ and } sub2 = 0 \Rightarrow (R_S, R_C) \geq M \tag{9}$$

$$sub2 = 1 \Rightarrow (R_S, R_C) \geq 2{\cdot}M \tag{10}$$

The optimized MSB-first algorithm illustrated in figure 3 compares the intermediate result (R_S, R_C) to $2{\cdot}2^n$ and 2^n instead of the actual values $2{\cdot}M$ and M. For this reason, it is possible that the intermediate result is not always fully reduced. But if the comparisons are performed according to the presented redundant number estimations, the algorithm guarantees that the intermediate result is always smaller than three times the modulus (i.e., smaller than $3{\cdot}2^{n-1}$) before the next multiplier bit $B[i]$ is processed. This is valid for any modulus M which satisfies equation (5), even for $M = M_{min}$.

After each addition of a partial-product, the modulo reduction is accomplished by continued subtractions of $2{\cdot}M$ and M. Of course this raises the question how many subtractions of $2{\cdot}M$ and/or M will be (at most) necessary. Because the redundant number estimation guarantees that the intermediate result (R_S, R_C) is smaller than $3{\cdot}2^{n-1}$ before the quantity $2{\cdot}(R_S, R_C) + A{\cdot}B[i]$ is computed, the product $2{\cdot}(R_S, R_C)$ is always smaller than $6{\cdot}2^{n-1}$. Since the partial-product $A{\cdot}B[i]$ is smaller than 2^n it is proven that the intermediate result is smaller than $8{\cdot}2^{n-1}$ before beginning the modulo reduction. Thus, for any modulus M satisfying equation (5), at most three subtractions of $2{\cdot}M$ or M are necessary until (R_S, R_C) is smaller than $3{\cdot}2^{n-1}$. On the other hand, a more precise quotient evaluation would reduce the number of subtractions. However, the proposed method benefits from the fact that the redundant number estimation does not cause a significant critical path and that no multiples of M need to be precomputed and stored.

4 Arithmetic in Binary Extension Fields GF(2^m)

The elements of GF(2^m) are polynomials of degree less than m, with coefficients in GF(2). For example, if $a(t)$ is an element in GF(2^m), then one can have

$$a(t) = \sum_{i=0}^{m-1} a_i\, t^i = a_{m-1}\, t^{m-1} + \ldots + a_2\, t^2 + a_1\, t + a_0 \quad \text{with} \quad a_i \in \{0, 1\} \tag{11}$$

This binary polynomial can also be written in bit-string form as $A[m-1\,..\,0]$, whereby $A[i]$ corresponds to the coefficient a_i. Finite fields of characteristic 2 are attractive for hardware implementation due to their “carry-free” arithmetic. The addition in GF(2^m) is implemented as component-wise exclusive OR (XOR), whilst the implementation of the multiplication depends on the *basis* chosen [14].

INPUT: An irreducible polynomial $p(t)$ of degree m, a multiplicand-polynomial $a(t)$, and a multiplier-polynomial $b(t)$.
OUTPUT: Result-polynomial $r(t) = a(t) \cdot b(t) \bmod p(t)$.

$r(t) \leftarrow 0$
for i **from** $m-1$ **downto** 0 **do**
    $r(t) \leftarrow t \cdot r(t) + a(t) \cdot b_i$
    if degree$(r(t)) = m$ **then** $r(t) \leftarrow r(t) - p(t)$
endfor

Fig. 4: MSB-first iterative multiplication in GF(2^m).

The simplest representation is in polynomial basis, where the multiplication is performed modulo an *irreducible polynomial* of degree exactly m.
A bit-serial polynomial basis multiplier for GF(2^m) has an area complexity of $\mathcal{O}(m)$ and computes a multiplication in m clock cycles. They have been well known since the early 1970s due to their exploration in coding theory [22], and later they have also been proposed for use in cryptography [2]. A recent publication reports a bit-serial architecture which is able to perform additions and multiplications over a variety of binary fields up to an order of 2^m [7].

4.1 Addition

The addition in GF(2^m) is performed by adding the coefficients modulo 2, which is nothing else than bit-wise XOR-ing the coefficients of equal powers of t. Compared to the addition of integers, the addition in GF(2^m) is much easier as it does not cause carry propagation. It is well known that in the field GF(2^m) any element $a(t)$ is its own additive inverse since $a(t) + a(t) = 0$, the additive identity. Consequently, addition and subtraction are equivalent operations in GF(2^m).

4.2 Multiplication

Multiplication in GF(2^m) involves multiplying the two polynomials together (carry-free coefficient multiplication) and then finding the residue modulo a given irreducible polynomial $p(t)$. In general, the reduction modulo an irreducible polynomial $p(t)$ requires polynomial division. For an efficient implementation it is necessary to perform the field multiplication without polynomial division. One possibility is to interleave the reduction modulo $p(t)$ with the multiplication operation, instead of performing the reduction separately after the multiplication of the polynomials is finished. This leads to a characteristic 2 version of the shift-and-add method, where the multiplication is realized by addition of partial-products, and the reduction is performed by subtraction of the irreducible polynomial. The pseudocode illustrated in figure 4 describes this algorithm.

The multiplication of two polynomials $a(t), b(t) \in$ GF(2^m) modulo an irreducible polynomial $p(t)$ is done by scanning the coefficients of the multiplier-polynomial $b(t)$ from b_{m-1} to b_0 and adding the partial-product $a(t) \cdot b_i$ to the intermediate result $r(t)$. The partial-product $a(t) \cdot b_i$ is either 0 (if $b_i = 0$) or the multiplicand-polynomial $a(t)$ (if $b_i = 1$). After each partial-product addition, the intermediate result must be multiplied by t to align it for the next partial-product. The reduction modulo the irreducible polynomial $p(t)$ is interleaved with the partial-product additions by subtraction of $p(t)$ if the degree of the intermediate result is m, i.e., if the coefficient r_m is 1. It turns out that the computation of $r(t) = a(t) \cdot b(t) \bmod p(t)$ requires m steps, and at each step we perform the following operations:

– computation of $t \cdot r(t)$ (a 1-bit left-shift)
– generation of a partial-product (logical AND between b_i and $a(t)$)
– addition of the partial-product (an $(m+1)$-bit XOR operation)
– generation of the subtrahend (logical AND between r_m and $p(t)$)
– subtraction of the subtrahend (an $(m+1)$-bit XOR operation)

The required logical operations are AND, XOR, and 1-bit left-shifts, which makes a hardware implementation of this algorithm very straightforward.

5 Multiplier Architecture

When taking a closer look at the multiplication algorithms for GF(p) (figure 3) and for GF(2^m) (figure 4), it is easily observed that these algorithms have some similarities. In both algorithms, one operand (the multiplier) is scheduled bit by bit, beginning with the MSB, and the other operand (the multiplicand) is scheduled fully parallel. Both algorithms perform three basic operations: Addition of partial-products, 1-bit left-shifts of the intermediate result, and subtraction(s) of the modulus (or the irreducible polynomial, respectively). The main difference is the way how the addition or subtraction is performed. An addition in GF(p) involves addition of integers and can be performed by carry-save adders, using a redundant representation for the result. On the other hand, the addition in GF(2^m) is a simple logical XOR operation.

5.1 Implementation of the Field Arithmetic

Figure 5 illustrates an arithmetic unit for implementation of the field additions and subtractions, respectively. All carry-save additions have to be performed with $(n+1)$-bit precision. The sum output R_S and the carry output R_C of the adders are latched on each half-cycle for synchronization purposes. Note that the circuit for generation of the partial-product as well as the circuit for generation of the subtrahend are not shown in figure 5.

A subtraction is usually performed by adding the two's complement of the subtrahend S, which can be realized in our case by addition of the bitwise complement of S and setting the initial carry Cin to 1. Therefore, addition and subtraction are essentially the same operation. It must be emphasized that the MSB-first algorithm from figure 3 guarantees that the intermediate result will never become negative, i.e., the $Cout$ output of the carry-save adders can be

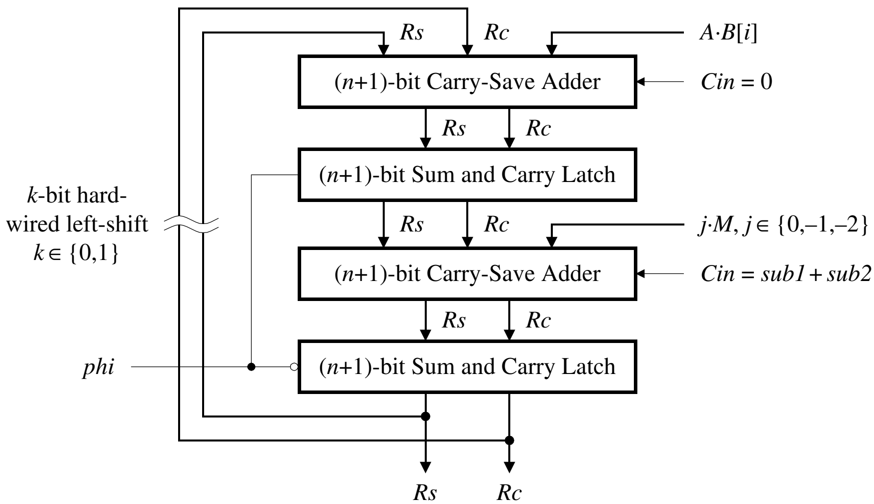

Fig. 5: Arithmetic unit of an n-bit unified multiplier.

ignored. In the following we describe how this arithmetic unit can be used to implement a modulo multiplication. Later in this section it will be shown that the arithmetic unit can also perform the addition/subtraction in GF(2^m).

According to the MSB-first iterative algorithm for modulo multiplication, the processing of a multiplier bit $B[i]$ takes place in the following way: The first carry-save adder at the top of figure 5 performs the addition of the partial-product $A{\cdot}B[i]$ to the current intermediate result. The sum output R_S and the carry output R_C of the first CSA are used to estimate the multiple of the modulus to become subtracted at the second CSA. This estimation is performed as described in section 3.1, and only the two highest order bits of R_S and R_C are needed to implement the logical functions of equation (6) and (7). Therefore, the hardware to decide whether to subtract $0{\cdot}M$, $1{\cdot}M$ or $2{\cdot}M$ can be implemented very efficiently and will not cause a long critical path in the arithmetic unit.

The subtraction of M or $2{\cdot}M$ is realized by addition of the two's complement of M or $2{\cdot}M$ to the output of the first CSA, which takes place at the second CSA. But one subtraction of $2{\cdot}M$ or M may not be enough to guarantee that the intermediate result is within the range of $[0, 3{\cdot}2^{n-1})$. Therefore, a control signal *xsub* is generated according to equation (6) in order to decide whether or not an extra subtraction of M or $2{\cdot}M$ is necessary. If an extra subtraction is required, the outputs R_S and R_C of the second CSA are fed back to the first (upper) CSA (without a left-shift). For an extra subtraction, the multiplier bit $B[i]$ must be masked off, so that no partial-product (i.e., zero) is added at the first CSA. After that, the extra subtraction of M or $2{\cdot}M$ takes place again at the second CSA.

If no extra subtraction is required, the processing of the multiplier bit is finished. The outputs of the second CSA are fed back to the inputs of the first CSA with a 1-bit hardwired left-shift. R_S and R_C are now correctly aligned for addition of the next partial-product and the same procedure starts again. After the last multiplier bit has been processed, the sum and carry of the second CSA represent the redundant result (R_S, R_C) of the modulo multiplication.

Generation of the partial-product. The partial-product $A \cdot B[i]$ is either 0 (if $B[i]$ is 0), or the multiplicand A (if $B[i]$ is 1). Thus, the generation of the partial-product $A \cdot B[i]$ is simply done by a bit-wise AND operation between the multiplier bit $B[i]$ and all the bits of the multiplicand A.

Generation of the subtrahend. The subtrahend $S = j \cdot M, j \in \{0, -1, -2\}$ must be generated according to the requirements of the optimized MSB-first algorithm. In the presented arithmetic unit the subtraction of S is realized by addition of the bitwise complement of S and by setting the initial carry Cin of the CSA to 1. The control signals *sub1* and *sub2* introduced in section 3.1 indicate whether the subtrahend S has to be 0, M, or $2 \cdot M$, and they can be used for generating the subtrahend-bits $S[i]$ according to the following equations: [1]

$$S[i] = sub1 \cdot \overline{sub2} \cdot \overline{M[i]} + sub2 \cdot \overline{M[i-1]} \quad \text{for} \quad i = 1 \dots n \tag{12}$$

$$S[0] = sub1 \cdot \overline{sub2} \cdot \overline{M[0]} + sub2 \tag{13}$$

Performing addition/subtraction in GF(2^m). The sum bit $R_S[i]$ of a full-adder calculates the logical XOR of its three inputs (see equation (3)). By setting all carry bits of the adders to 0, the sum outputs $R_S[i]$ of the adders provide the functionality of a 2-input XOR gate. This is exactly the functionality required for addition/subtraction in GF(2^m). Also the partial-products are generated in exactly the same way as described before, namely by a logical AND of the coefficient $B[i]$ and all the coefficients[2] of the multiplicand polynomial $a(t)$. A reduction of the intermediate result is necessary whenever the degree of the result-polynomial is m, i.e., if $R_S[m]$ is 1. The requirement for a subtraction of the irreducible polynomial $p(t)$ is indicated by the control signal *sub1*, since $sub1 = R_S[m]$ if the carry bits $R_C[i]$ are set to 0:

$$sub1 = R_S[m] + 0 + (R_S[m-1] \cdot 0) = R_S[m] \quad \text{and} \quad sub2 = R_S[m] \cdot 0 = 0$$

The control signal *sub2* is always 0. As mentioned in subsection 4.2, the generation of the subtrahend S is a logical AND between the control signal *sub1* and the bits of the irreducible polynomial, i.e., $S[i] = sub1 \cdot P[i] = R_S[m] \cdot P[i]$. The presented arithmetic unit provides exactly the functionality required for the multiplication in the binary extension field GF(2^m) when the carry bits $R_C[i]$ are set to 0.

---

[1] The algorithm also works with the following control signals: $sub1 = R_S[n] \otimes R_C[n]$, $sub2 = R_S[n] \cdot R_C[n]$, and $xsub = R_S[n] + R_C[n] + (R_S[n-1] \cdot R_C[n-1])$. In this case the generation of the subtrahend bits $S[i]$ is simplified to the following equation: $S[i] = sub1 \cdot \overline{M[i]} + sub2 \cdot \overline{M[i-1]}$.

[2] According to the bit-string notation introduced in section 4.1, the coefficient x_i of a polynomial $x(t)$ is denoted as $X[i]$.

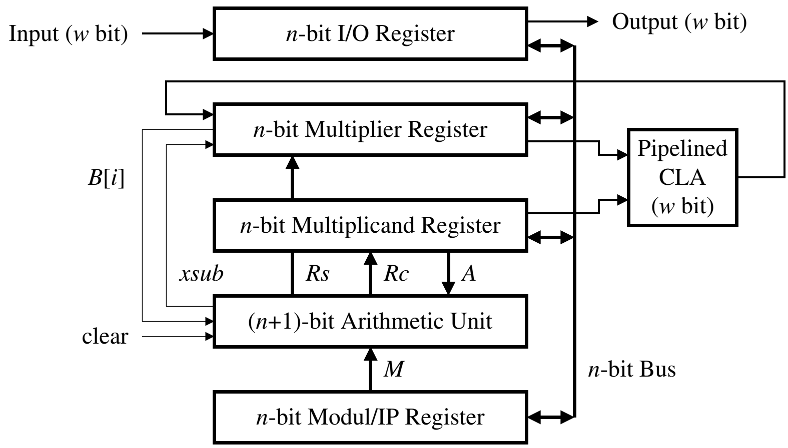

Fig. 6: Block diagram of the bit-serial multiplier architecture.

5.2 The Unified Multiplier Architecture

Figure 6 shows the bit-serial multiplier architecture, consisting of the $(n+1)$-bit arithmetic unit, four n-bit registers, and a pipelined w-bit carry-lookahead adder [21], whereby w denotes the wordsize of the registers (usually 8, 16, or 32 bits). The *I/O Register* performs data transfers from and to the world outside the multiplier. We prefer to provide a seperate register for I/O operations to ensure that the overall performance of the multiplier is not reduced by slow data transfers. The *Modulus/IP Register* is needed to store the bit-string representation of the modulus or the irreducible polynomial, respectively. *Multiplicand* and *Multiplier Register* are used for storing the current operands of the multiplication. Both registers can carry out w-bit shift operations in LSB direction, register *Multiplier* can additionally perform 1-bit shift operations in MSB direction (1-bit left-shifts). All four registers are connected through an n-bit bus.

After the operands have been loaded into the corresponding registers, a modulo multiplication takes place in the following way: The *Multiplier* register is shifted bit by bit in MSB direction to deliver the multiplier bits $B[n-1]$ to $B[0]$ to the arithmetic unit. The processing of the multiplier bits $B[i]$ is performed as described in subsection 5.1. The control signal *xsub* is generated from the two most significant bits of R_S and R_C of the second CSA according to equation (6). Whenever *xsub* is 1, the arithmetic unit has to perform an extra subtraction and register *Multiplier* must stop the left-shift until $xsub = 0$. After the least significant multiplier bit $B[0]$ has been processed, the redundant result (R_S, R_C) is loaded into registers *Multiplier* and *Multiplicand*, respectively. Note that the old values of the multiplier and multiplicand are not needed any more. Now the redundant result must be converted into non-redundant representation. This is done by the pipelined w-bit carry-lookahead adder (CLA) and requires $\lceil n/w \rceil$ clock cycles plus the delay of the CLA (usually $\log_2(w)$ clock cycles). The out-

Table 2: Typical subtraction sequences of $2 \cdot M$ and/or M depending on the range of (R_S, R_C) after addition of the partial-product.

Range of (R_S, R_C)	Typical subtraction sequence	Clock cycles
$0 \leq (R_S, R_C) < 2^{n-1}$	—	1
$2^{n-1} \leq (R_S, R_C) < 2 \cdot 2^{n-1}$	—	1
$2 \cdot 2^{n-1} \leq (R_S, R_C) < 3 \cdot 2^{n-1}$	—	1
$3 \cdot 2^{n-1} \leq (R_S, R_C) < 4 \cdot 2^{n-1}$	M	1
$4 \cdot 2^{n-1} \leq (R_S, R_C) < 5 \cdot 2^{n-1}$	$2 \cdot M$	1
$5 \cdot 2^{n-1} \leq (R_S, R_C) < 6 \cdot 2^{n-1}$	$2 \cdot M, M$	2
$6 \cdot 2^{n-1} \leq (R_S, R_C) < 7 \cdot 2^{n-1}$	$2 \cdot M, 2 \cdot M$	2
$7 \cdot 2^{n-1} \leq (R_S, R_C) < 8 \cdot 2^{n-1}$	$2 \cdot M, 2 \cdot M, M$	3

put of the CLA is fed back to the *Multiplier* register. Since the non-redundant result may not be in the range of $[0, 2^n)$, an additional modulus subtraction and redundant to non-redundant conversion may be necessary. After the modulo multiplication has finished, the result resides within register *Multiplier*.

In GF(2^m)-mode, a multiplication is performed in a similar way, except that no extra subtractions and no redundant to non-redundant conversions of the result are necessary.

5.3 Performance Estimation

Since the carry-save adders are separated by latches, the addition of a partial-product and the first subtraction of M or $2 \cdot M$ are performed in one clock cycle. Any extra modulus subtraction requires an additional clock cycle. As stated in subsection 3.1, at most three subtractions of $2 \cdot M$ and/or M are necessary to guarantee that the intermediate result is smaller than $3 \cdot M$ (i.e., $3 \cdot 2^{n-1}$). Therefore, the processing of a single multiplier bit takes at most three clock cycles. The actual number of cycles depends on the size of the intermediate result *after* addition of the partial-product. Table 2 shows typical subtraction sequences depending on the range of the intermediate result (R_S, R_C). The values at the third column represent the number of clock cycles required for partial-product addition *and* the subtractions. For example, if $6 \cdot 2^{n-1} \leq (R_S, R_C) < 7 \cdot 2^{n-1}$ then typically two subtractions have to be performed. Consequently, two clock cycles are necessary for the processing of that multiplier bit.

According to the subtraction sequences shown in table 2, one can assume that any bit of the multiplier takes on average 1.5 clock cycles to be processed. When given arbitrary n-bit operands, the computation of $A \cdot B$ mod M requires approximately $1.5 \cdot n$ clock cycles. Moreover, one or two redundant to non-redundant conversions of the result are necessary, each needs $\lceil n/w \rceil + \log_2(w)$ clock cycles. For an $(n+1)$-bit arithmetic unit and a w-bit CLA, the number of clock cycles

Table 3: Principal operation characteristics of the unified multiplier.

Operation	Integer		GF(p)		GF(2^m)	
	add.	mult.	add.	mult.	add.	mult.
Cycles per bit	–	1	–	≈ 1.5	–	1
Max. op. length	$n-1$	$n/2$	n	n	n	n
Operand align	right	right	left	left	left	left

for a modulo multiplication can be estimated as follows:

$$c \approx 1.5 \cdot n + 1.5 \cdot \left(\left\lceil \frac{n}{w} \right\rceil + \log_2(w) \right) \approx 1.5 \cdot n \tag{14}$$

This means that a multiplication in the prime field GF(p) requires approximately $1.5 \cdot \lceil \log_2(p) \rceil$ clock cycles. On the other hand, a multiplication in GF(2^m) is finished after m cycles since any bit of the multiplier takes exactly one clock cycle to be processed.

6 Summary of Results and Conclusions

The subject of this paper was to present a novel bit-serial multiplier architecture which operates over finite fields GF(p) and GF(2^m). A multiplication in GF(p) is performed in a serial/parallel manner, which means that the multiplier is scheduled sequentially (bit by bit) and the multiplicand is scheduled fully parallel. The modulo reduction is interleaved with the multiplication by subtractions of once or twice the modulus. Thus, the arithmetic unit has to perform only three simple operations: Addition of partial-products, left-shift of the intermediate result, and subtraction of once or twice the modulus. Compared to other bit-serial multipliers, the proposed architecture profits from an efficient subtrahend estimation circuit which does not cause a significant critical path. The modulo multiplier described in this paper is also capable to perform multiplications in GF(2^m), i.e., it is a unified (dual-field) multiplier for GF(p) and GF(2^m). Contrary to architectures which use Montgomery multiplication, the introduced MSB-first algorithm requires neither operand transformation into Montgomery domain nor precomputed constants.

The presented design is scalable in size, and an n-bit multiplier operates over a wide range of finite fields. For example, a multiplier dimensioned for 200 bits can also be used for fields of smaller order, like 192 or 163 bits, by left-aligning all operands in the registers. Furthermore, the multiplier can also perform ordinary integer addition and multiplication, respectively. The operand size for ordinary integer multiplication is limited to about $n/2$ bits since the product can't exceed n-bit precision. Table 3 summarizes principal characteristics of addition and multiplication over integers, prime fields GF(p), and binary extension fields GF(2^m).

The unified multiplier can be implemented for an area-cost only slightly higher than that of the multiplier for the prime field GF(p), providing significant area savings when both types of multiplier are needed. To be more specific, the overhead introduced by the dual-field arithmetic is just a logic circuit for setting the carry bits of the CSA to 0, which means that this feature comes almost for free. Additionally, the architecture is neither restricted to use primes of a special form (e.g., generalized Mersenne primes), nor does it favor particular irreducible polynomials like trinomials or pentanomials. Another advantage of the bit-serial architecture is its high degree of regularity. The presented unified multiplier offers a fair area/performance trade-off, which makes it attractive for the implementation of a crypto-coprocessor for low-end 8-bit smart cards.

The correctness of the presented concepts was verified by a functional, cycle-based model of the multiplier architecture written is a hardware description language. Our future work will be a VLSI implementation of the multiplier.

References

1. P. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko (ed.), *Advances in Cryptology — CRYPTO '86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 311–323. Springer-Verlag, Berlin, Germany, 1987.
2. T. Beth, B. M. Cook, and D. Gollmann. Architectures for exponentiation in GF(2^n). In A. M. Odlyzko, (ed.), *Advances in Cryptology — CRYPTO '86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 302–310. Springer-Verlag, Berlin, Germany, 1987.
3. I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*, vol. 265 of *London Mathematial Society Lecture Notes Series*. Cambridge University Press, Cambridge, UK, 1999.
4. G. R. Blakley. A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.
5. E. F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In D. Chaum, R. L. Rivest, and A. T. Sherman (eds.), *Advances in Cryptology: Proceedings of CRYPTO '82*, pp. 51–60. Plenum Press, New York, NY, USA, 1982.
6. J. Goodman and A. Chandrakasan. An energy efficient reconfigurable publik-key cryptography processor architecture. In Ç. K. Koç and C. Paar (eds.), *Cryptographic Hardware and Embedded Systems — CHES 2000*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 174–191. Springer-Verlag, Berlin, Germany, 2000.
7. J. Großschädl. A low-power bit-serial multiplier for finite fields GF(2^m). In *Proceedings of the 34th IEEE International Symposium on Circuits and Systems (ISCAS 2001)*, vol. IV, pp. 37–40, 2001.
8. Y.-J. Jeong and W. P. Burleson. VLSI array algorithms and architectures for RSA modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(2):211–217, June 1997.
9. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, January 1987.
10. N. Koblitz, A. J. Menezes, and S. A. Vanstone. The state of elliptic curve cryptography. *Designs, Codes and Cryptography*, 19(2/3):173–193, March 2000.
11. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

12. P. Kornerup. High-radix modular multiplication for cryptosystems. In G. Jullien, M. J. Irwin, and E. E. Swartzlander (eds.), *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pp. 277–283. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.
13. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. In H. Imai and Y. Zheng (eds.), *Public Key Cryptography — PKC 2000*, vol. 1751 of *Lecture Notes in Computer Science*, pp. 446–465. Springer-Verlag, Berlin, Germany, 2000.
14. R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Second edition. Cambridge University Press, Cambridge, UK, 1994.
15. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*, vol. 234 of *The Kluwer International Series in Engineering and Computer Science.* Kluwer Academic Publishers, Boston, MA, USA, 1993.
16. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams (ed.), *Advances in Cryptology — CRYPTO '85*, vol. 218 of *Lecture Notes in Computer Science*, pp. 417–426. Springer-Verlag, Berlin, Germany, 1986.
17. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
18. H. Morita. A fast modular-multiplication algorithm based on a higher radix. In G. Brassard (ed.), *Advances in Cryptology — CRYPTO '89*, vol. 435 of *Lecture Notes in Computer Science*, pp. 387–399. Springer-Verlag, Berlin, Germany, 1990.
19. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Federal Information Processing Standards (FIPS) Publication 186-2. Online available at `http://csrc.nist.gov/encryption`. February 2000.
20. H. Orup, E. Svendsen, and E. Andreasen. VICTOR an efficient RSA hardware implementation. In I. Damgård, (ed.), *Advances in Cryptology — EUROCRYPT '90*, vol. 473 of *Lecture Notes in Computer Science*, pp. 245–252. Springer-Verlag, Berlin, Germany, 1991.
21. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, New York, NY, USA, 2000.
22. W. W. Peterson and E. J. Weldon. *Error-Correcting Codes.* Second edition. MIT Press, Cambridge, MA, USA, 1972.
23. E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In Ç. K. Koç and C. Paar (eds.), *Cryptographic Hardware and Embedded Systems — CHES 2000*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 277–292. Springer-Verlag, Berlin, Germany, 2000.
24. H. Sedlak. The RSA cryptography processor. In D. Chaum and W. L. Price (eds.), *Advances in Cryptology — EUROCRYPT '87*, vol. 304 of *Lecture Notes in Computer Science*, pp. 95–105. Springer Verlag, Berlin, Germany, 1988.
25. N. Takagi. A radix-4 modular multiplication hardware algorithm for modular exponentiation. *IEEE Transactions on Computers*, 41(8):949–956, August 1992.
26. C. D. Walter. Faster modular multiplication by operand scaling. In J. Feigenbaum (ed.), *Advances in Cryptology — CRYPTO '91*, vol. 576 of *Lecture Notes in Computer Science*, pp. 313–323. Springer-Verlag, Berlin, Germany, 1992.