Acta Appl Math (2006) 93: 75–118
DOI 10.1007/s10440-006-9072-z

Efficient Hardware Implementation of Finite Fields with Applications to Cryptography

Jorge Guajardo · Tim Güneysu · Sandeep S. Kumar · Christof Paar · Jan Pelzl

Received: 30 August 2006 / Accepted: 30 August 2006 /
Published online: 26 September 2006

Abstract The paper presents a survey of most common hardware architectures for finite field arithmetic especially suitable for cryptographic applications. We discuss architectures for three types of finite fields and their special versions popularly used in cryptography: binary fields, prime fields and extension fields. We summarize algorithms and hardware architectures for finite field multiplication, squaring, addition/subtraction, and inversion for each of these fields. Since implementations in hardware can either focus on high-speed or on area-time efficiency, a careful choice of the appropriate set of architectures has to be made depending on the performance requirements and available area.

Key words Field arithmetic · cryptography · efficient implementation · binary field arithmetic · prime field arithmetic · extension field arithmetic · Optimal extension fields.

Mathematics Subject Classifications (2000) 12-02 · 12E30 · 12E10.

J. Guajardo (✉)
Information and System Security Department, Philips Research, Eindhoven, The Netherlands
e-mail: Jorge.Guajardo@philips.com

T. Güneysu · S. S. Kumar · C. Paar · J. Pelzl
Horst-Görtz Institute for IT-Security, Ruhr-University Bochum, Germany

T. Güneysu
e-mail: gueneysu@crypto.rub.de

S. S. Kumar
e-mail: kumar@crypto.rub.de

C. Paar
e-mail: cpaar@crypto.rub.de

J. Pelzl
e-mail: pelzl@crypto.rub.de

1 Introduction

Before 1976, Galois fields and their hardware implementation received considerable attention because of their applications in coding theory and the implementation of error correcting codes. In 1976, Diffie and Hellman [20] invented public-key cryptography[1] and single-handedly revolutionized a field which, until then, had been the domain of intelligence agencies and secret government organizations. In addition to solving the key management problem and allowing for digital signatures, public-key cryptography provided a major application area for finite fields. In particular, the Diffie-Hellman key exchange is based on the difficulty of the Discrete Logarithm (DL) problem in finite fields. It is apparent, however, that most of the work on arithmetic architectures for finite fields only appeared after the introduction of two public-key cryptosystems based on finite fields: elliptic curve cryptosystems (ECC), introduced by Miller and Koblitz [39, 47], and hyperelliptic cryptosystems (HECC), a generalization of elliptic curves introduced by Koblitz in [40].

Both, prime fields and extension fields, have been proposed for use in such cryptographic systems but until a few years ago the focus of hardware implementations was mainly on fields of characteristic 2. This is due to two main reasons. First, even characteristic fields naturally offer a straight forward manner in which field elements can be represented. In particular, elements of $\mathbb{F}_2$ can be represented by the logical values '0' and '1' and thus, elements of $\mathbb{F}_{2^m}$ can be represented as vectors of 0s and 1s. Second, until 1997 applications of fields $\mathbb{F}_{p^m}$ for odd p were scarce in the cryptographic literature. This changed with the appearance of works such as [13, 14, 41, 46, 64] and more recently with the introduction of pairing-based cryptographic schemes [5]. The situation with prime fields $\mathbb{F}_p$ has been slightly different as the type of arithmetic necessary to support for these fields is in essence the same as that needed for the RSA cryptosystem [60], the most widely used cryptosystem to this day, and the Digital Signature Algorithm [52].

The importance of the study of architectures for finite fields lies on the fact that a major portion of the runtime of cryptographic algorithms is spent on finite field arithmetic computations. For example, in the case of the asymmetric cryptosystems such as RSA, DSA, or ECC, most time is spent on the computation of modular multiplications. Therefore performance gains of such core routines directly affects the performance of the entire cryptosystem. In addition, although various efficient algorithms exist for finite field arithmetic for signal processing applications, the algorithms suitable for practical cryptographic implementations vary due to the relatively large size of finite field operands used in cryptographic applications. A single public-key encryption such as, an RSA or a DSA operation can involve thousands of modular multiplications with 1,024-bit long or larger. Especially in hardware, many degrees of freedom exist during the implementation of a cryptographic system and this demands for a careful choice of basic building blocks, adapted to one's needs.

[1] The discovery of public-key cryptography in the intelligence community is attributed in [23] to John H. Ellis in 1970. The discovery of the equivalent of the RSA cryptosystem [60] is attributed to Clifford Cocks in 1973 while the equivalent of the Diffie–Hellman key exchange was discovered by Malcolm J. Williamson, in 1974. However, it is believed (although the issue remains open) that these British scientists did not realize the practical implications of their discoveries at the time of their publication within CESG (see for example [21, 62]).

Furthermore, there are stricter constraints which need to be fulfilled concerning a target platform like smart-cards and RFID tags, where tight restrictions in terms of minimal energy consumption and area must be met.

We present here a survey of different hardware architectures for the three different types of finite fields that are most commonly used in cryptography:

– Prime fields $\mathbb{F}_p$,
– Binary fields $\mathbb{F}_{2^m}$, and
– Extension fields $\mathbb{F}_{p^m}$ for odd primes p.

The remainder of the paper is organized as follows. In Section 2, we describe architectures for $\mathbb{F}_p$ fields. These fields are probably the most widely used in cryptographic applications. In addition, they constitute basic building blocks for architectures supporting $\mathbb{F}_{p^m}$ field operations where p is an odd prime. In Section 3, we introduce the representation of extension field ($\mathbb{F}_{2^m}$ and $\mathbb{F}_{p^m}$) elements used in this paper. Sections 4 and 5 describe hardware implementation architectures for $\mathbb{F}_{2^m}$ and $\mathbb{F}_{p^m}$ fields, respectively. Section 7 concludes this contribution.

2 Hardware Implementation Techniques over $\mathbb{F}_p$

In this section, we survey hardware architectures for performing addition, subtraction, multiplication, and inverse in $\mathbb{F}_p$ fields, where p is an odd prime. Section 2.1 deals with integer adders which will be fundamental building blocks for the $\mathbb{F}_p$ multipliers presented in Section 2.2.

2.1 Addition and Subtraction in $\mathbb{F}_p$

It is well known that adders constitute the basic building blocks for more complicated arithmetic operators such as multipliers. Thus, this section surveys adder architectures which will be used in the next sections to implement more complicated operators. For more detailed treatments of hardware architectures and computer arithmetic, we refer the reader to [42, 55].

In what follows, we consider the addition of two n-bit integers $A = \sum_{i=0}^{n-1} a_i 2^i$ and $B = \sum_{i=0}^{n-1} b_i 2^i$, with $S = c_{out} 2^n + \sum_{i=0}^{n-1} s_i 2^i = A + B$ being possibly an $(n+1)$-bit integer. We refer to A and B as the inputs (and their bits a_i and b_i as the input bits) and S as the sum (and its bits s_i for $i = 0 \cdots n-1$ as the sum bits). The last bit of the sum c_{out} receives the special name of carry-out bit.

In the case of modular addition and subtraction, the result of an addition or subtraction has to be reduced. Generally, in the case of an addition we check whether the intermediate result $A + B \geq p$ (where p is the modulus) and eventually reduce the result by subtracting the modulus p once. In the case of subtraction, we usually check whether $A - B < 0$ and eventually add the modulus.

In the following, we will describe architectures for a simple addition and subtraction circuit. For the modular arithmetic, some control logic for reducing the intermediate results has to be added.

2.1.1 Building Blocks for Adders and Subtracters

Single-bit half-adders (HA) and full-adders (FA) are the basic building blocks used to synthesize more complex adders. A HA accepts two input bits a and b and outputs a sum-bit s and a carry-out bit c_{out} following Eqs. (1) and (2)

$$s = a \oplus b \tag{1}$$

$$c_{out} = a \wedge b \tag{2}$$

A half-adder can be seen as a single-bit binary adder that produces the 2-bit binary sum of its inputs, i.e., $a + b = (c_{out}\ s)_2$. In a similar manner, a full-adder accepts a 3-bit input a, b and a carry-in bit c_{in}, and outputs 2 bits: a sum-bit s and a carry-out bit c_{out}, according to Eqs. (3) and (4)

$$s = a \oplus b \oplus c_{in} \tag{3}$$

$$\begin{aligned} c_{out} &= (a \wedge b) \vee (c_{in} \wedge (a \vee b)) \\ &= (a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in}) \end{aligned} \tag{4}$$

Pictorially, we can view half-adders and full-adders as depicted in Figures 1 and 2.

In the next paragraph, we will discuss the simplest version of an adder and how adders can be used for subtraction.

2.1.2 Ripple-Carry Adders (RCA)

An n-bit ripple-carry adder (RCA) can be synthesized by concatenating n single-bit FA cells, with the carry-out bit of the ith-cell used as the carry-in bit of the $(i + 1)$th-cell. The resulting n-bit adder outputs an n-bit long sum and a carry-out bit.

Addition and subtraction usually is implemented as a single circuit. A subtraction $x - y$ can simply be computed by the addition of x, $\overline{y}$, and 1, where $\overline{y}$ is the bitwise complement of y. Hence, the subtraction can rely on the hardware for addition.

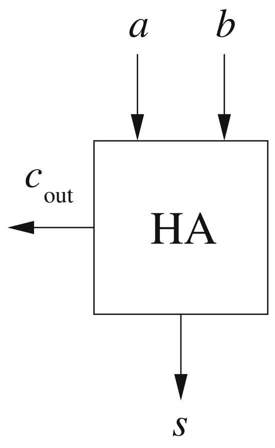


Figure 1 Half-adder cell.

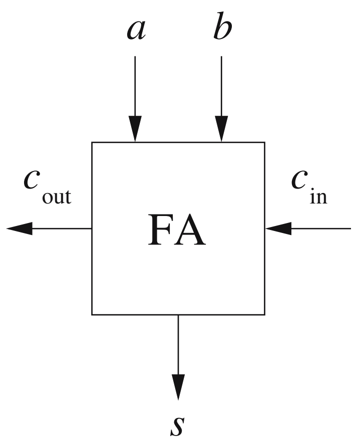


Figure 2 Full-adder cell.

Figure 3 shows a combined addition and subtraction circuit where only one input bit has to be changed (`sub=1`) in order to compute a subtraction rather than an addition.

The total latency of an n-bit RCA can be approximated by $n \cdot T_{\mathrm{FA}}$, where T_{FA} refers to the delay of a single full-adder cell. Designing n-bit RCAs for any value of n is a rather simple task: simply, concatenate as many FA cells as bits of precision are required. In addition, although not directly relevant to the treatment here, RCA-based designs have two other advantages: (a) easy sign detection if one uses 2's complement arithmetic, and (b) subtraction is accomplished by first converting the subtrahend to its 2's complement representation and then adding the result to the original minuend. However, the delay of the RCA grows linearly with n, making it undesirable for large values of n or for high-speed applications, as it is the case in many cryptographic systems. Thus, the need to explore other designs to improve the performance of the adder without significantly increasing area-resource requirements.

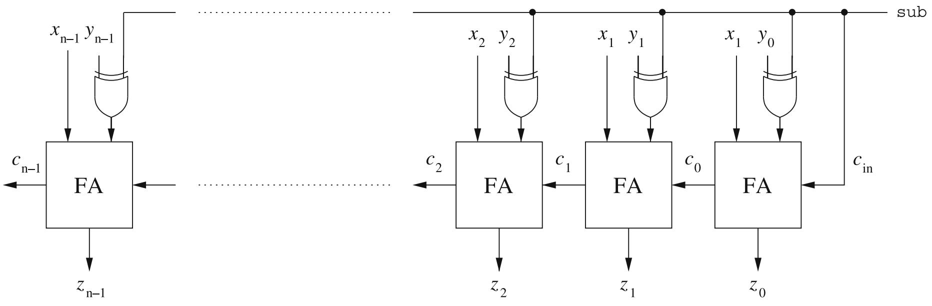


Figure 3 Simple addition and subtraction circuit based on an n-bit RCA.

2.1.3 Carry-Lookahead Adders (CLA)

As its name indicates a carry lookahead adder (CLA) computes the carries generated during an addition before the addition process takes place, thus, reducing the total time delay of the RCA at the cost of additional logic. In order to compute carries ahead of time we define next generate, propagate, and annihilate signals.

Definition 1 Let a_i, b_i be two operand digits in radix-r notation and c_i be the carry-in digit. Then, we define the generate signal g_i, the propagate signal p_i, and the annihilate (absorb) signal v_i as:

$$\begin{aligned} g_i &= 1 \quad \text{if} \quad a_i + b_i \geq r \\ p_i &= 1 \quad \text{if} \quad a_i + b_i = r - 1 \\ v_i &= 1 \quad \text{if} \quad a_i + b_i < r - 1 \end{aligned}$$

where $c_i, g_i, p_i, v_i \in \{0, 1\}$ and $0 \leq a_i, b_i < r$.

Notice that Definition 1 is independent of the radix used which allows one to treat the carry propagation problem independently of the number system [55]. Specializing to the binary case and using the signals from Definition 1, we can re-write g_i, p_i, and v_i as:

$$g_i = a_i \wedge b_i \tag{5}$$

$$p_i = a_i \oplus b_i \tag{6}$$

$$v_i = \overline{a_i} \wedge \overline{b_i} = \overline{a_i \vee b_i} \tag{7}$$

Relations (5), (6), and (7) have very simple interpretations. If $a_i, b_i \in GF(2)$, then a carry will be generated whenever both a_i and b_i are equal to 1, a carry will be propagated if either a_i or b_i are equal to 1, and a carry will be absorbed whenever both input bits are equal to 0. In some cases it is also useful to define a transfer signal ($t_i = a_i \vee b_i$) which denotes the event that the carry-out will be 1 given that the carry-in is equal to 1.[2] Combining Eqs. (4), (5), and (6) we can write the carry-recurrence relation as follows:

$$c_{i+1} = g_i \vee (c_i \wedge t_i) = g_i \vee (c_i \wedge p_i) \tag{8}$$

Intuitively, Eq. (8) says that there will be a non-zero carry at stage $i + 1$ either if the generate signal is equal to 1 or there was a carry at stage i and it was propagated (or transferred) by this stage. Notice that implementing the carry-recurrence using the transfer signal leads to slightly faster adders than using the propagate signal, since

[2] Notice that different authors use different definitions. We have followed the definitions of [55], however [42] only defines two types of signals a generate signal, which is the same as the generate signal from [55], and a propagate signal which is equivalent to [55] transfer signal. The resulting carry recurrence relations are nevertheless the same.

an OR gate is easier to produce than an XOR gate [55]. Notice that from Eqs. (3) and (6), it follows that

$$s_i = p_i \oplus c_i \tag{9}$$

Thus, Eqs. (9) and (8) define the CLA.

2.1.4 Carry-Save Adders (CSA)

A CSA is simply a parallel ensemble of n full-adders without any horizontal connection, i.e., the carry bit from adder i is not fed to adder $i+1$ but rather, stored as c'_i. In particular given three n-bit integers $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, and $C = \sum_{i=0}^{n-1} c_i 2^i$, their sum produces two integers $C' = \sum_{i=0}^{n} c'_i 2^i$ and $S = \sum_{i=0}^{n-1} s_i 2^i$ such that

$$C' + S = A + B + C$$

where:

$$s_i = a_i \oplus b_i \oplus c_i \tag{10}$$

$$c'_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i) \tag{11}$$

where $c'_0 = 0$ (notice that Eqs. (10) and (11) are nothing else but Eqs. (1) and (2) re-written for different inputs and outputs). An n-bit CSA is shown in Figure 4.

We point out that since the inputs A, B, and C are all applied in parallel the total delay of a CSA is equal to that of a full-adder cell (i.e., the delay of Eqs. (10) and (11)). On the other hand, the area of the CSA is just n-times the area of an FA cell and it scales very easily by adding more FA-cells in parallel. Subtraction can be accomplished by using 2's complement representation of the inputs.

However, CSAs have two major drawbacks:

– Sign detection is hard. In other words, when an integer is represented as a carry-save pair (C', S) such that its actual value is $C' + S$, we may not know the sign of the total sum $C' + S$ unless the addition is performed in full length. In [35] a method for fast sign estimation is introduced and applied to the construction of modular multipliers.

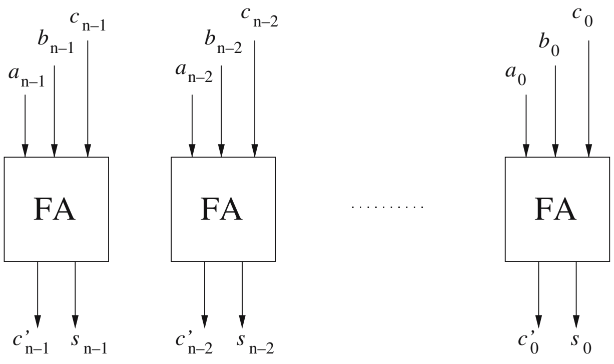


Figure 4 n-bit carry-save adder.

– CSAs do not solve the problem of adding two integers and producing a single output. Instead, it adds three integers and produces two outputs.

2.1.5 Carry-Delayed Adders (CDA)

Carry-delayed adders (CDAs) were originally introduced in [53] as a modification to the CSA paradigm. In particular, a CDA is a two-level CSA. Thus, adding $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, and $C = \sum_{i=0}^{n-1} c_i 2^i$, we obtain the sum-pair (D, T), such that $D + T = A + B + C$, where

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_i \\ c'_{i+1} &= (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i) \\ t_i &= s_i \oplus c'_i \qquad (12) \\ d_{i+1} &= s_i \wedge c'_i \qquad (13) \end{aligned}$$

with $c'_0 = d_0 = 0$. Notice that Eqs. (12) and (13) are exactly the same equations that define a half-adder, thus an n-bit CDA is nothing else but an n-bit CSA plus a row of n half-adders. The overall latency is equal to the delay of a full-adder and a half-adder cascaded in series, whereas the total area is equal to n times the area of a full-adder and a half adder. The CDA scales in same manner as the CSA.

2.1.6 Summary and Comparison

We described four different integer adders: ripple-carry adders, carry-lookahead adders, carry-save adders, and carry-delayed adders. The asymptotic complexity of the above adders is summarized in Table 1 and it is well known.

2.2 Multiplication in $\mathbb{F}_p$

As in the case of modulo adders, we have also divided the multipliers according to the method of implementation. Thus, we have modulo multipliers based on adders, table-lookups, hybrid architectures, and purely combinatorial circuits.

We will describe hardware architectures of two algorithmic concepts for the modular multiplication. The modular Montgomery multiplication and the interleaved

Table 1 Asymptotic area/time complexities of different n-bit adders

Adder type	Abbreviation	Area	Time
Ripple-carry adder	RCA	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Carry-lookahead adder	CLA	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$
Carry-save adder	CSA	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Carry-delayed adder	CDA	$\mathcal{O}(n)$	$\mathcal{O}(1)$

modular multiplication allow for an area-time efficient design. Modular multiplication in $\mathbb{F}_p$ is the mathematical operation

$$x \cdot y \bmod p$$

with $y, x, p \in \mathbb{F}_p$ and $x,\ y < p$, whereby x and y are called the operands and p denotes the modulus. In current practical cryptologic applications, x, y and p are large numbers of 100 bit and above. There exists many different algorithms for modular multiplication. All these algorithms belong to one of two groups:

- *Parallel algorithms*: most such algorithms are time optimal and calculate the result with a time complexity of $O(\log p)$ [74]. Their disadvantage is a huge area complexity, resulting in an expensive hardware implementation. But many practical applications require low-cost solutions, especially now where an increasing number of high volume products require cryptographic foundations (e.g., in consumer electronics).
- *Sequential algorithms*: the sequential algorithms of highest importance are the classical modular multiplication [36], Barrett modular multiplication [4], interleaved modular multiplication [11, 63], and modular Montgomery multiplication [50].

In the classical modular multiplication, operands are multiplied and the result is divided by modulus. The remainder of this division is the result of the modular multiplication. The disadvantage of the classical modular multiplication is the size of the intermediate result, which is twice the size of the operands and more importantly, its area and time complexities. Barrett replaces the modular multiplication by three standard multiplications and some additions. The disadvantage of this solution is the high time complexity of three multiplications. During interleaved modular multiplication the multiplication and the calculation of the remainder of the division are interleaved. The advantage is that the length of the intermediate result is only 1 or 2 bits larger than the operands. The disadvantage is the use of subtractions in order to reduce the intermediate results. More detailed information about interleaved modular multiplication will be provided in the following. The modular Montgomery multiplication is the most frequently used algorithm for modular multiplication. The computation is done in the *Montgomery domain*. As an advantage of this computation, we do not need subtractions to reduce the intermediate results. The disadvantage is the fact that the modular Montgomery multiplication calculates

$$x \cdot y \cdot 2^{-k} \pmod p \qquad \text{instead of} \qquad x \cdot y \pmod p.$$

Hence, two modular Montgomery multiplications are required for one modular multiplication. More detailed information about modular Montgomery multiplication is given in following paragraphs.

2.2.1 Interleaved Modular Multiplication

The idea of interleaved modular multiplication is quite simple: the first operand is multiplied with the second operand in a bitwise manner and added to the intermediate result. The intermediate result is reduced with respect to the modulus.

Algorithm 1 Interleaved Modular Multiplication

```
Input: $x, y, p$ with $0 \le x, y \le p$
Output: $u = x \cdot y \bmod p$
    $k$: number of bits of $x$
    $x_i$: $i^{\text{th}}$ bit of $x$
 1: $u = 0$;
 2: for $i = k - 1; i \ge 0; i = i - 1$ do
 3:         $u = 2 \cdot u$;
 4:         $v = x_i \cdot y$;
 5:         $u = u + v$;
 6:     if $u \ge p$ then
 7:         $u = u - p$;
 8:     end if
 9:     if $u \ge p$ then
10:         $u = u - p$;
11:     end if
12: end for
```

For this purpose two subtractions per iteration are required. Algorithm 1 provides a pseudo code description of interleaved modular multiplication. Algorithm 1 has several drawbacks: the first problem is the comparison of u and p in Steps 6 and 9. In the worst case all bits of p and u must be compared. This problem can be solved by an approximate comparison with 2^k instead of p. The second problem is the number of additions or subtractions in Steps 5, 7, and 10. All these operations can be replaced by one addition only. In order to do so, we estimate the number of times p should be subtracted, and find out if y will be added in the next loop iteration. There are only few possible values for this estimation. These can be precomputed and stored in a look-up table. In each loop iteration, the estimation of the previous iteration can be added to the intermediate result. Figure 5 depicts an optimized architecture of the interleaved modular multiplication.

The advantage of this version is the fact that only one addition per iteration of the loop is required. It is also possible to use a CSA in order to obtain a higher clock frequency, as suggested by [15].

2.2.2 Montgomery Modular Multiplication

Montgomery modular multiplication is based on the same concept as the interleaved modular multiplication: the first operand is multiplied bitwise with the second operand. The results of these partial multiplications are added successively from the least significant to the most significant bit differently from interleaved modular multiplication. In each iteration, we determine whether the intermediate result is odd or even. For this purpose the least significant bit of the intermediate result is inspected. In case this bit is equal to ‘1,’ the modulus is added to the intermediate sum. This guarantees the sum to be even. The intermediate result is then divided by 2. Algorithm 2 describes the Montgomery modular multiplication.

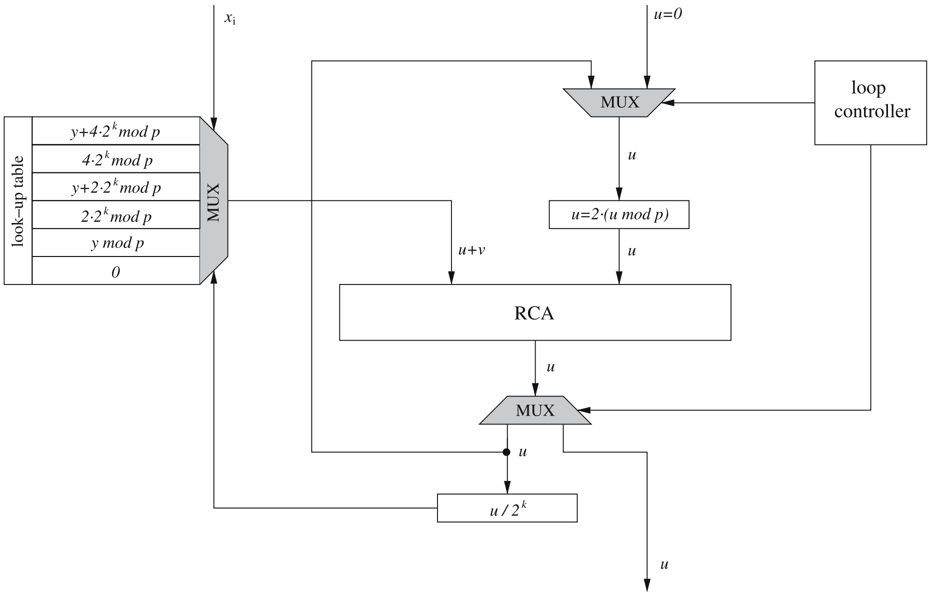


Figure 5 Interleaved modular multiplication with RCA.

A modular multiplication for numbers in standard representation requires two modular Montgomery multiplications:

$$\begin{aligned}\hat{z} &= \mathit{Montgomery}(x, y, p)\\ &= x \cdot y \cdot 2^{-k} \bmod p\\ &\quad \text{and}\\ z &= \mathit{Montgomery}(\hat{z}, 2^{2k} \bmod p, p)\\ &= (x \cdot y \cdot 2^{-k} \bmod p) \cdot (2^{2k} \bmod p) \cdot 2^{-k} \bmod p\\ &= (x \cdot y \cdot 2^{-k} \cdot 2^{2k} \cdot 2^{-k}) \bmod p\\ &= x \cdot y \bmod p.\end{aligned}$$

The advantage of Algorithm 2 is that it does not require any subtractions. However, the algorithm requires two additions per loop iteration and these additions are slow because they are non-redundant. This problem was solved an [16] where a very efficient architecture was presented for Montgomery multiplication with CSA. This architecture is shown in Figure 6.

A detailed comparison of efficient hardware architectures for Montgomery multiplication and interleaved multiplication can be found in [3].

Algorithm 2 Montgomery Modular Multiplication

```
Input: $x, y < p < 2^k$, with $2^{k-1} < p < 2^k$ and $p = 2t + 1$, with $t \in \mathbb{N}$.
Output: $u = x \cdot y \cdot 2^{-k} \bmod p$.
    $k$: number of bit in $x$,
    $x_i$: $i^{\text{th}}$ bit of $x$
 1: $u = 0$;
 2: for $i = 0; i < k; i++$ do
 3:     $u = u + x_i \cdot y$
 4:     if $u_0 = 1$ then
 5:         $u = u + p$;
 6:     end if
 7:     $u = u$ div 2;
 8: end for
 9: if $u \geq p$ then
10:     $u = u - p$;
11: end if
```

2.3 Architectures for Small Moduli

2.3.1 Table Look-up and Hybrid-based Architectures

The naive method to implement modular multiplication via table look-ups would require $m^2\lceil\log_2(m)\rceil$ bits of storage with $m = \lceil p \rceil$. Early implementations based on table look-ups can be found in [33, 67]. However, several techniques have been developed to improve on these memory requirements.

2.3.2 Combinatorial Architectures

This section considers modulo multipliers based on combinational logic for *fixed* moduli. We emphasize that only architectures for fixed moduli have been considered. In addition, it would not be fair to compare architectures which can process multiple moduli to architectures optimized for a single modulus. To our knowledge, the best architectures for variable moduli in the context of RNS is the one presented in [17]. In Di Claudio et al. [17] introduced the pseudo-RNS representation. This new representation is similar in flavor to the Montgomery multiplication technique as it defines an auxiliary modulus A relatively-prime to p. The technique allows building reprogrammable modulo multipliers, systolization, and simplifies the computation of DSP algorithms. Nevertheless, ROM-based solutions seem to be more efficient for small moduli $p < 2^6$ [17].

In Soudris et al. [66] present full-adder (FA) based architectures for RNS multiply-add operations which adopt the carry-save paradigm. The paper concludes that for moduli $p > 2^5$, FA based solutions outperform ROM ones. Finally, [57] introduces a new design which takes advantage of the non-occurring combinations of input bits to reduce certain 1-bit FAs to OR gates and, thus, reduce the overall area complexity of the multiplier. The multiplier outperforms all previous designs in terms of area. However, in terms of time complexity, the designs in [17, 61] as well as ROM-based ones outperform the multiplier proposed in [57] for most prime moduli $p < 2^7$.

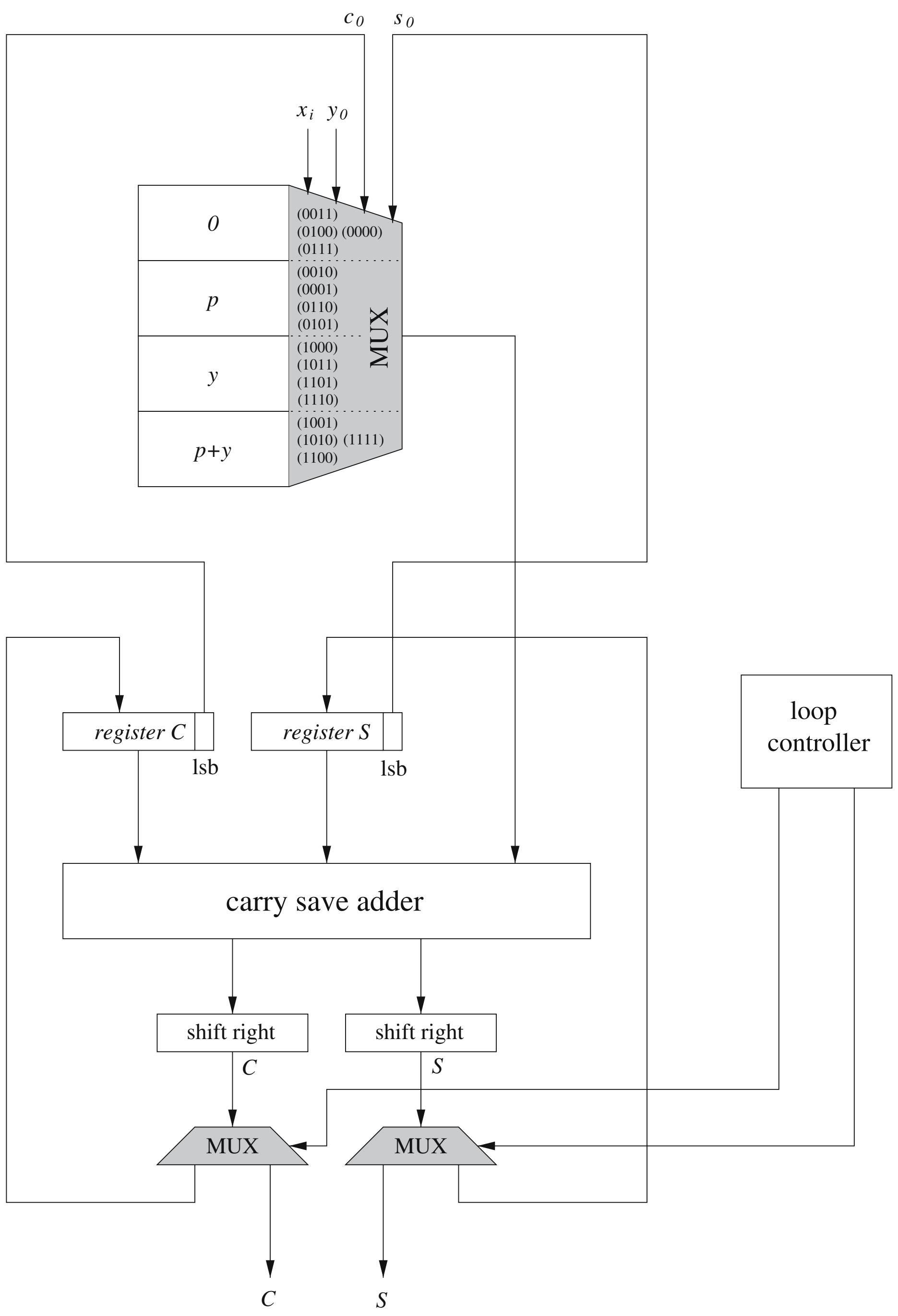


Figure 6 Montgomery modular multiplication with one CSA.

Nevertheless, the combined time/area product in [57] is always less than that of other designs.

2.4 Inversion in $\mathbb{F}_p$

Several cryptographic methods based on prime fields $\mathbb{F}_p$ require an operation to absorb the effect of a modular multiplication and hence find for a given value a, a number $b = a^{-1} \bmod p$ which fulfills the congruence $a \cdot b \equiv 1 \bmod p$. To find such a corresponding number b, we basically have two directions for an implementation. The first is based on exploiting Fermat's little theorem which translates into modular exponentiation and the second relies on the principle of the extended Euclidean (GCD) algorithm. Unfortunately, both approaches have significant disadvantages, hence a lot of efforts have been taken to avoid or combine modular inversions as often as possible by algorithmic modifications. In Elliptic Curve Cryptography for example, the projective point representation is usually preferred for hardware application because it trades inversions for multiplications which simplifies a design enormously in terms of area costs.

2.4.1 Fermat's Little Theorem

From Fermat's little theorem we know that $a^{p-1} \equiv 1 \bmod p$. With this theorem, the modular inversion can be easily implemented by computing $b = a^{-1} \equiv a^{p-2} \bmod p$. This obviously can be implemented in hardware by an exponentiation fundamentally relying on a clever choice of an underlying modular multiplication method (c.f. Section 2.2) which is repetitively applied. The original multiplication hardware is augmented by an exponentiation controller either implementing a classical binary method or using a more advanced windowing technique. The smarter methods, however, usually require additional precomputations and, thus, additional memory to store the precomputed values. Figure 7 shows the basic schematic of a simple Fermat based binary exponentiation circuit.

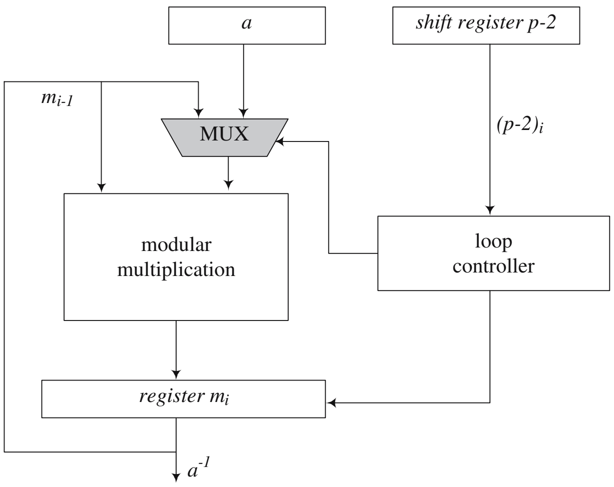


Figure 7 Finite field inversion over $\mathbb{F}_p$ using exponentiation hardware.

The drawback of this type of approach is the slow execution speed for a single inversion. In particular, exponentiation has an overall complexity of $\mathcal{O}(n^3)$. A hardware implementation which employs the Fermat based method has an area complexity of $\mathcal{O}(n^2)$ and a time complexity of $\mathcal{O}(n^2)$, respectively. This is often unacceptable. Hence, further attempts to reduce the execution time of an inversion will be shown in the next section.

2.4.2 Extended GCD Algorithms

Another approach to inversion is the implementation of the extended Euclidean algorithm (EEA) which computes the modular inverse of a number $a \in \mathbb{F}_p$ by finding two variables b, q that satisfy the relation

$$ab + pq = \gcd(a, p) = 1$$

Computing mod p on both sides of the equation, the term pq vanishes and the inverse of a is finally obtained as b. The operation of the EEA is based on the iterative computation of the $\gcd(a, m)$ with corresponding updates of the intermediate combinations of the coefficients leading to b, q. Due to the costly divisions in hardware which are required to compute the GCD, binary variants have been developed which are more appropriate for an implementation in hardware. Additionally, the Kaliski algorithm which is based on a two step computation of such a binary EEA variant is capable to perform an inversion between standard integer and Montgomery domain [49].

2.4.3 Binary Extended Euclidean Algorithm (BEA)

The Binary Extended Euclidean Algorithm was first proposed by R. Silver and J. Tersian in 1962 and it was published by G. Stein in 1967 [37]. It trades the divisions performed in the original EEA method for bit shifts which are much better suited for hardware applications. Hence, the algorithm can be efficiently implemented using just adder and subtracter circuits in combination with shift registers for the division. This benefit in the reduction of hardware complexity is at the expense of an increased number of iterations. Here, the upper bound for the iteration count turns out to be $2(\lfloor \log_2 x \rfloor + \lfloor \log_2 y \rfloor + 2)$ [51]. Although several authors have proposed efficient hardware architectures and implementations for the BEA [69, 71], more recent research has concentrated on the development of Montgomery based inversion which will be discussed in greater detail in the next section. Modular arithmetic in the Montgomery domain over $\mathbb{F}_p$ always has the great advantage of a straightforward modular reduction instead of a costly division.

2.4.4 Kaliski Inversion for Montgomery Domain

The modular inverse for Montgomery arithmetic was first introduced by Kaliski in 1995 [34] as a modification of the extended binary GCD algorithm. This method provides some degree of computational freedom by finding the modular inverse in two phases: First, for a given value $a \in \mathbb{F}_p$ in standard representation or $a \cdot 2^m \in \mathbb{F}_p$ in the Montgomery domain, an almost modular inverse r with $r = a^{-1} \cdot 2^z \bmod p$ is computed. Secondly, the output r is corrected by a second phase which reduces the

Table 2 Configurations of the Kaliski inversion for different domains

Domain	After phase I	Phase II operation	Total complexity
Standard → Standard	$r = a^{-1} \cdot 2^z$	$r = (r + p \cdot r_0)/2$	$2z$
Standard → Montgomery	$r = a^{-1} \cdot 2^z$	$r = (r + p \cdot r_0)/2$	$2z - m$
Montgomery → Standard	$r = a^{-1} \cdot 2^{z-h}$	$r = 2(r - p \cdot r_0)$	$2z - m$
Montgomery → Montgomery	$r = a^{-1} \cdot 2^{z-h}$	$r = 2(r - p \cdot r_0)$	$2m$

exponent z in r either to $z = 0$ or $z = m$ of the target domain. Thus, the algorithm can be used for several combinations for converting values from and to the Montgomery domain. Table 2 shows the possible options and their corresponding necessary iteration count and the required type of operation for the correction performed in the second phase. Note that due to the n-bit modulus p with $p > a$, the exponent z is always in the range $n < z < 2n$. Hence, in the worst case $4n - 2$ iterations have to be performed for determining the modular inverse in standard domain whereas the inverse in Montgomery representation solely requires about $2m$ where m denotes the Montgomery radix.

From the sequential structure of Algorithm 3 it becomes clear that a direct transfer to a hardware architecture suffers from a long critical path due to inner conditions as well as the necessity for several parallel arithmetic components. Thus, the inversion is costly in terms of area with a relatively low maximum clocking speed compared

Algorithm 3 Almost Montgomery Inverse (AMI)

```
Input: $a \in \mathbb{F}_p$
Output: $r$ and $z$ where $r = a^{-1} \cdot 2^z \bmod p$ and $m \le z \le 2m$
 1: $u \leftarrow p, v \leftarrow a, r \leftarrow 0, s \leftarrow 1$
 2: $k \leftarrow 0$
 3: while $v > 0$ do
 4:     if $u$ is even then
 5:         $u \leftarrow u/2, s \leftarrow 2s$
 6:     else if $v$ is even then
 7:         $v \leftarrow v/2, r \leftarrow 2r$
 8:     else if $u > v$ then
 9:         $u \leftarrow (u - v)/2, r \leftarrow r + s, s \leftarrow 2s$
10:     else
11:         $v \leftarrow (v - u)/2, s \leftarrow r + s, r \leftarrow 2r$
12:     end if
13:     $k \leftarrow k + 1$
14: end while
15: if $r \ge p$ then
16:     $r \leftarrow r - p$                {Make sure that $r$ is within its boundaries}
17: end if
18: return $r \leftarrow p - r$
```

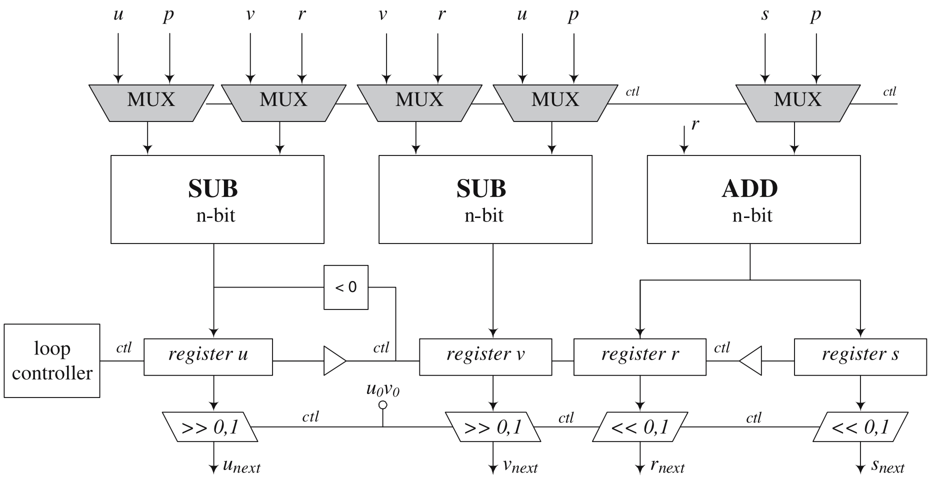


Figure 8 Basic Kaliski inversion design.

to the proposed architectures for multiplication and addition in previous sections. A basic inverter design is shown in Figure 8 based on one n-bit adder and two n-bit subtracters.

There are several improvements to minimize hardware requirements and signal latency of the Kaliski inverter due to the long carry propagation path in the n-bit wide adders and subtracters. An efficient VLSI architecture has been described in [29], whereas in [22] the total critical path is reduced by using $n/2$-bit arithmetics which enables a higher clocking frequency and increases the overall throughput. An area optimized inversion design, however, was reported in [10] by Bucik and Lorencz. The authors modified the AMI algorithm by introducing 2's-complement number representation which relaxed the critical path dependencies and allowed for a design using only a single n-bit adder and subtracter.

3 Extension Fields $\mathbb{F}_{2^m}$ and $\mathbb{F}_{p^m}$: Preliminaries

3.1 Basis Representation

For the discussion that follows, it is important to point out that there are several possibilities to represent elements of extension fields. Thus, in general, given an irreducible polynomial $F(x)$ of degree m over $\mathbb{F}_q$ and a root α of $F(x)$ (i.e., $F(\alpha) = 0$), one can represent an element $A \in \mathbb{F}_{q^m}$, $q = p^n$ and p prime, as a polynomial in α, i.e., as $A = a_{m-1}\alpha^{m-1} + a_{m-2}\alpha^{m-2} + \cdots + a_1\alpha + a_0$ with $a_i \in \mathbb{F}_q$. The set $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is then said to be a polynomial basis (or standard basis) for the finite field $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$. Another type of basis is called a normal basis. Normal bases are of the form $\{\beta, \beta^q, \beta^{q^2}, \ldots, \beta^{q^{m-1}}\}$ for an appropriate element $\beta \in \mathbb{F}_{q^m}$. Then, an element $B \in \mathbb{F}_{q^m}$ can be represented as $B = b_{m-1}\beta^{q^{m-1}} + b_{m-2}\beta^{q^{m-2}} + \cdots + b_1\beta^q + b_0\beta$ where $b_i \in \mathbb{F}_q$. It can be shown that for any field $\mathbb{F}_q$ and any extension field $\mathbb{F}_{q^m}$, there exists always a normal basis of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$(see [43, Theorem 2.35]). Notice that $(\beta^{q^i})^{q^k} = \beta^{q^{i+k}} =$

$\beta^{q^{i+k \bmod m}}$ which follows from the fact that $\beta^{q^m} \equiv \beta$ (i.e., Fermat's little theorem). Thus, raising an element $B \in \mathbb{F}_{q^m}$ to the qth power can be easily accomplished through a cyclic shift of its coordinates, i.e., $B^q = (b_{m-1}\beta^{q^{m-1}} + b_{m-2}\beta^{q^{m-2}} + \cdots + b_1\beta^q + b_0\beta)^q = b_{m-2}\beta^{q^{m-1}} + b_{m-3}\beta^{q^{m-2}} + \cdots + b_0\beta^q + b_{m-1}\beta$, where we have used the fact that in any field of characteristic p, $(x+y)^q = x^q + y^q$, where $q = p^n$. Finally, the dual basis has also received attention in the literature. Two bases $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ and $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ are said to be dual or complementary bases if for $0 \le i, j \le m-1$ we have:

$$\mathrm{Tr}_{E/F}(\alpha_i\beta_j) = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases}$$

A variation on dual bases is introduced in [48, 75] where the concept of a weakly dual basis is defined. As a final remark notice that given a basis $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$, one can always represent an element $\beta \in \mathbb{F}_{q^m}$ as:

$$\beta = b_0\alpha_0 + b_1\alpha_1 + \cdots + b_{m-1}\alpha_{m-1}$$

where $b_i \in \mathbb{F}_q$.

3.2 Notation

In Sections 4 and 5 we describe hardware implementation techniques for fields $\mathbb{F}_{2^m}$ and $\mathbb{F}_{p^m}$, where p is odd, respectively. Thus, in general we can speak of the field $\mathbb{F}_{p^m}$, where $p = 2$ in the case of $\mathbb{F}_{2^m}$ and odd otherwise. The field is generated by an irreducible polynomial $F(x) = x^m + G(x) = x^m + \sum_{i=0}^{m-1} g_i x^i$ over $\mathbb{F}_p$ of degree m. We assume α to be a root of $F(x)$, thus for $A, B, C \in \mathbb{F}_{p^m}$, we write $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$, $C = \sum_{i=0}^{m-1} c_i\alpha^i$, and $a_i, b_i, c_i \in \mathbb{F}_p$. Notice that by assumption $F(\alpha) = 0$ since α is a root of $F(x)$. Therefore,

$$\alpha^m = -G(\alpha) = \sum_{i=0}^{m-1} -g_i\alpha^i \tag{14}$$

gives an easy way to perform modulo reduction whenever we encounter powers of α greater than $m-1$. Eq. (14) reduces to

$$\alpha^m = G(\alpha) = \sum_{i=0}^{m-1} g_i\alpha^i \tag{15}$$

for fields of characteristic 2. Addition in $\mathbb{F}_{p^m}$ can be achieved as shown in Eq. (16)

$$C(\alpha) \equiv A(\alpha) + B(\alpha) = \sum_{i=0}^{m-1} (a_i + b_i)\alpha^i \tag{16}$$

where the addition $a_i + b_i$ is done in $\mathbb{F}_p$, (e.g. $a_i \in \{0, 1\}$ for $\mathbb{F}_{2^m}$). Multiplication of two elements $A, B \in \mathbb{F}_{p^m}$ is written as $C(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i \equiv A(\alpha) \cdot B(\alpha)$, where the multiplication is understood to happen in the finite field $\mathbb{F}_{p^m}$ and all α^t, with $t \ge m$ can be reduced using Eq. (14). Notice that we abuse our notation and throughout the text we will write $A \bmod F(\alpha)$ to mean *explicitly* the reduction step described previously. Finally, we refer to A as the multiplicand and to B as the multiplier.

4 Hardware Implementation Techniques for Fields $\mathbb{F}_{2^m}$

Characteristic 2 fields $\mathbb{F}_{2^m}$ are often chosen for hardware realizations [12] as they are well suited for hardware implementation due to their ‘carry-free’ arithmetic. This not only simplifies the architecture but reduces the area due to the lack of carry arithmetic. For hardware implementations trinomial and pentanomial reduction polynomials are chosen as they enable a very efficient implementation. We present here efficient architectures for multiplier and squarer implementations for binary fields in hardware. The inversion architecture is similar in design to that used in the extension fields of odd characteristic described in Section 6 and is therefore not discussed here.

4.1 Multiplication in $\mathbb{F}_{2^m}$

Multiplication of two elements $A, B \in \mathbb{F}_{2^m}$, with $A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i$ and $B(\alpha) = \sum_{i=0}^{m-1} b_i\alpha^i$ is given as

$$C(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i \equiv A(\alpha) \cdot B(\alpha) \bmod F(\alpha)$$

where the multiplication is a polynomial multiplication, and all α^t, with $t \geq m$ are reduced with Eq. (15). The simplest algorithm for field multiplication is the shift-and-add method [37] with the reduction step inter-leaved shown as Algorithm 4.

Notice that in Step 3 of Algorithm 4 the computation of $b_i A$ and $C\alpha \bmod F(\alpha)$ can be performed in parallel as they are independent of each other. However, the value of C in each iteration depends on both the value of C at the previous iteration and on the value of $b_i A$. This dependency has the effect of making the MSB multiplier have a longer critical path than that of the Least Significant Bit (LSB) multiplier, described later in the next section.

For hardware, the shift-and-add method can be implemented efficiently and is suitable when area is constrained. When the bits of B are processed from the most-significant bit to the least-significant bit (as in Algorithm 4), then it receives the name of *Most-Significant Bit-serial (MSB)* multiplier [65].

Algorithm 4 Shift-and-Add Most Significant Bit (MSB) First $\mathbb{F}_{2^m}$ Multiplication

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$ where $a_i, b_i \in \mathbb{F}_2$.
Output: $C \equiv A \cdot B \bmod F(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$ where $c_i \in \mathbb{F}_2$.
1: $C \leftarrow 0$
2: for $i = m - 1$ downto 0 do
3:     $C \leftarrow C \cdot \alpha \bmod F(\alpha) + b_i \cdot A$
4: end for
5: Return ($C$)
```

4.1.1 Reduction mod $F(\alpha)$

In the MSB multiplier, a quantity of the form $W\alpha$, where $W(\alpha) = \sum_{i=0}^{m-1} w_i\alpha^i \in \mathbb{F}_{2^m}$, has to be reduced mod $F(\alpha)$. Multiplying W by α, we obtain

$$W\alpha = \sum_{i=0}^{m-1} w_i\alpha^{i+1} = w_{m-1}\alpha^m + \sum_{i=0}^{m-2} w_i\alpha^{i+1} \tag{17}$$

Using the property of the reduction polynomial as shown in Eq. (15), we can substitute for α^m and re-write the index of the second summation in Eq. (17). $W\alpha$ mod $F(\alpha)$ can then be calculated as follows:

$$W\alpha \bmod F(\alpha) = \sum_{i=0}^{m-1}(g_i w_{m-1})\alpha^i + \sum_{i=1}^{m-1} w_i\alpha^i = (g_0 w_{m-1}) + \sum_{i=1}^{m-1}(w_{i-1} + g_i w_{m-1})\alpha^i$$

where all coefficient arithmetic is done modulo 2. As an example, we consider the structure of a 163-bit MSB multiplier shown in Figure 9.

Here, the operand A is enabled onto the data-bus A of the multiplier directly from the memory register location. The individual bits of b_i are sent from a memory location by implementing the memory registers as a cyclic shift-register (with the output at the most-significant bit).

The reduction within the multiplier is performed on the accumulating result c_i, as in Step 3 in Algorithm 4. The taps that are fed back to c_i are based on the reduction polynomial. Figure 9 shows an implementation for the reduction polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$, where the taps XOR the result of c_{162} to c_7, c_6 c_3 and c_0. The complexity of the multiplier is n AND + $(n + t - 1)$ XOR gates and n FF where $t = 3$ for a trinomial reduction polynomial and $t = 5$ for a pentanomial reduction polynomial. The latency for the multiplier output is n clock cycles. The maximum critical path is $2\Delta_{XOR}$ (independent of n) where, Δ_{XOR} represents the delay in an XOR gate.

Similarly a *Least-Significant Bit-serial (LSB)* multiplier can be implemented and the choice between the two depends on the design architecture and goals. In an LSB multiplier, the coefficients of B are processed starting from the least significant bit b_0

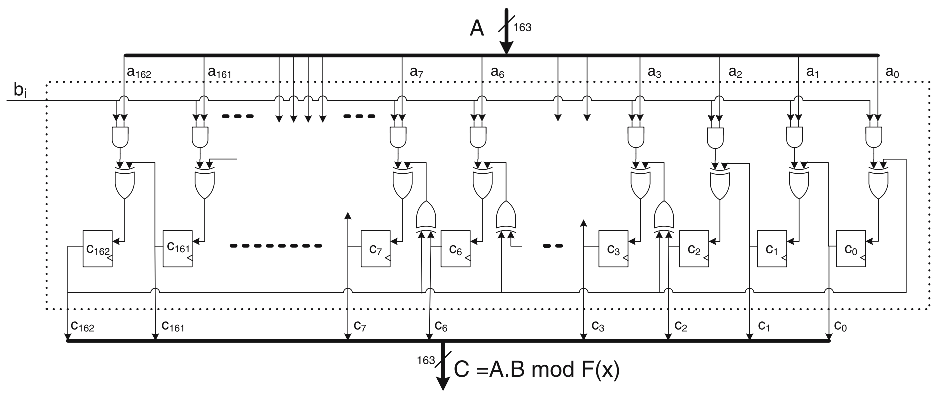


Figure 9 $\mathbb{F}_{2^{163}}$ Most significant bit-serial (MSB) multiplier circuit.

and continues with the remaining coefficients one at a time in ascending order. Thus multiplication according to this scheme is performed in the following way:

$$\begin{aligned}
C &\equiv AB \bmod F(\alpha) \\
&\equiv b_0A + b_1(A\alpha \bmod F(\alpha)) + b_2(A\alpha^2 \bmod F(\alpha)) \\
&\quad + \ldots + b_{m-1}(A\alpha^{m-1} \bmod F(\alpha)) \\
&\equiv b_0A + b_1(A\alpha \bmod F(\alpha)) + b_2((A\alpha)\alpha \bmod F(\alpha)) \\
&\quad + \ldots + b_{m-1}((A\alpha^{m-2})\alpha \bmod F(\alpha))
\end{aligned}$$

4.1.2 Digit Multipliers

Introduced by Song and Parhi in [65], they consider trade-offs between speed, area, and power consumption. This is achieved by processing several of B's coefficients at the same time. The number of coefficients that are processed in parallel is defined to be the digit-size D. The total number of digits in the polynomial of degree $m-1$ is given by $d = \lceil m/D \rceil$. Then, we can re-write the multiplier as $B = \sum_{i=0}^{d-1} B_i\alpha^{Di}$, where

$$B_i = \sum_{j=0}^{D-1} b_{Di+j}\alpha^j \ , \ 0 \leq i \leq d-1 \tag{18}$$

and we assume that B has been padded with zero coefficients such that $b_i = 0$ for $m-1 < i < d \cdot D$ (i.e., the size of B is $d \cdot D$ coefficients but $\deg(B) < m$). The multiplication can then be performed as:

$$C \equiv A \cdot B \bmod F(\alpha) = A \sum_{i=0}^{d-1} B_i\alpha^{Di} \bmod F(\alpha) \tag{19}$$

The *Least-Significant Digit-serial (LSD)* multiplier is a generalization of the LSB multiplier in which the digits of B are processed starting from the least significant to the most significant. Using Eq. (19), the product in this scheme can be computed as follows

$$\begin{aligned}
C &\equiv A \cdot B \bmod F(\alpha) \\
&\equiv \left[B_0A + B_1\left(A\alpha^D \bmod F(\alpha)\right) + B_2\left(A\alpha^D\alpha^D \bmod F(\alpha)\right)\right. \\
&\quad \left. + \ldots + B_{d-1}\left(A\alpha^{D(d-2)}\alpha^D \bmod F(\alpha)\right)\right] \bmod F(\alpha)
\end{aligned}$$

Algorithm 5 shows the details of the LSD Multiplier.

Remark 1 If C is initialized to value $I \in \mathbb{F}_{2^m}$ in Algorithm 5, then we can obtain as output the quantity, $A \cdot B + I \bmod F(\alpha)$ at no additional (hardware or delay) cost. This operation, known as a multiply/accumulate operation is very useful in elliptic curve based systems.

Algorithm 5 Least Significant Digit-serial (LSD) Multiplier [65]

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, where $a_i \in \mathbb{F}_2$, $B = \sum_{i=0}^{\lceil \frac{m}{D} \rceil - 1} B_i\alpha^{Di}$, where $B_i$ as in Eq. (18)
Output: : $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in \mathbb{F}_2$
1: $C \leftarrow 0$
2: for $i = 0$ to $\lceil \frac{m}{D} \rceil - 1$ do
3:     $C \leftarrow B_i A + C$
4:     $A \leftarrow A\alpha^D \bmod F(\alpha)$
5: end for
6: Return ($C \bmod F(\alpha)$)
```

4.1.3 Reduction mod $F(\alpha)$ *for Digit Multipliers*

In an LSD multiplier, products of the form $W\alpha^D \bmod F(\alpha)$ occur (as seen in Step 4 of Algorithm 5) which have to be reduced. As in the LSB multiplier case, one can derive equations for the modular reduction for *general* irreducible $F(\alpha)$ polynomials. However, it is more interesting to search for polynomials that minimize the complexity of the reduction operation. In coming up with these optimum irreducible polynomials we use two theorems from [65].

Theorem 1 [65] *Assume that the irreducible polynomial is of the form $F(\alpha) = \alpha^m + g_k\alpha^k + \sum_{j=0}^{k-1} g_j\alpha^j$, with $k < m$. For $t \leq m - 1 - k$, α^{m+t} can be reduced to degree less than m in one step with the following equation:*

$$\alpha^{m+t} \bmod F(\alpha) = g_k\alpha^{k+t} + \sum_{j=0}^{k-1} g_j\alpha^{j+t} \tag{20}$$

Theorem 2 [65] *For digit multipliers with digit-element size D, when $D \leq m - k$, the intermediate results in Algorithm 5 (Step 4 and Step 6) can be reduced to degree less than m in one step.*

Theorems 1 and 2, implicitly say that for a given irreducible polynomial $F(\alpha) = \alpha^m + g_k\alpha^k + \sum_{j=0}^{k-1} g_j\alpha^j$, the digit-element size D has to be chosen based on the value of k, the second highest degree in the irreducible polynomial.

The architecture of the LSD multiplier is shown in Figure 10 and consists of three main components.

1. The *main reduction circuit* to shift A left by D and reduce the result mod$F(\alpha)$ (Step 4 Algorithm 5).
2. The *multiplier core* which computes the intermediate C and stores it in the accumulator (Step 3 Algorithm 5).
3. The *final reduction circuit* to reduce the contents in the accumulator to get the final result C (Step 6 Algorithm 5).

All the components run in parallel requiring one clock cycle to complete each step and the critical path of the whole multiplier normally depends on the critical path of the multiplier core.

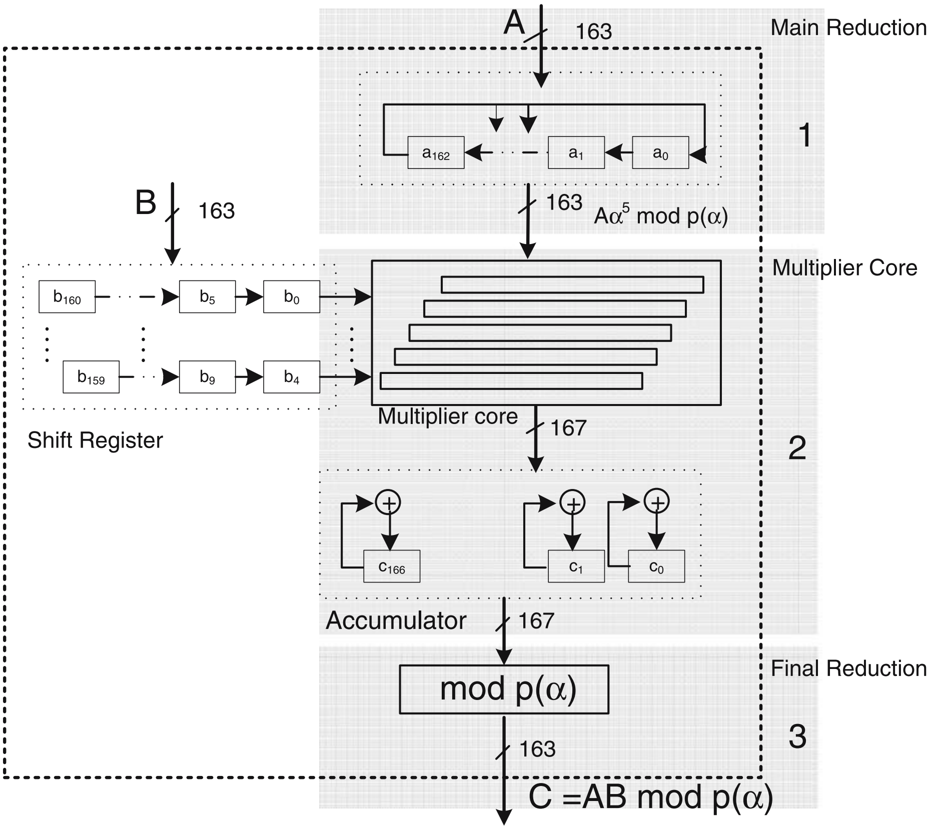


Figure 10 LSD-single accumulator multiplier architecture.

Figure 11 SAM core.

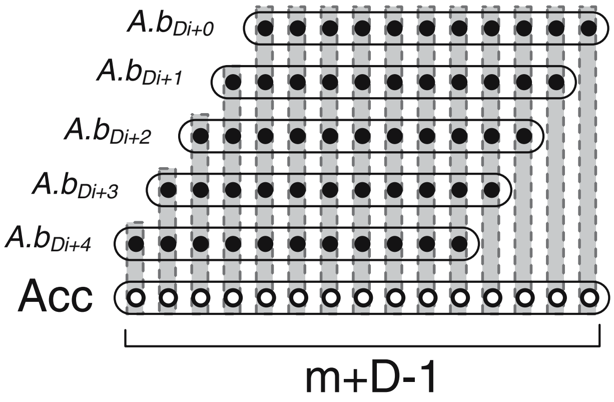
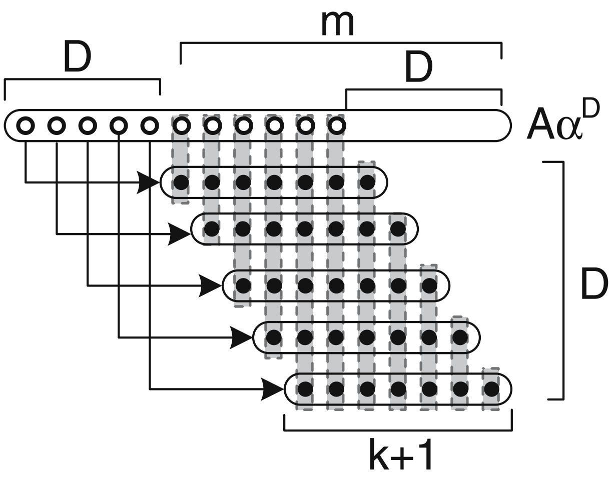
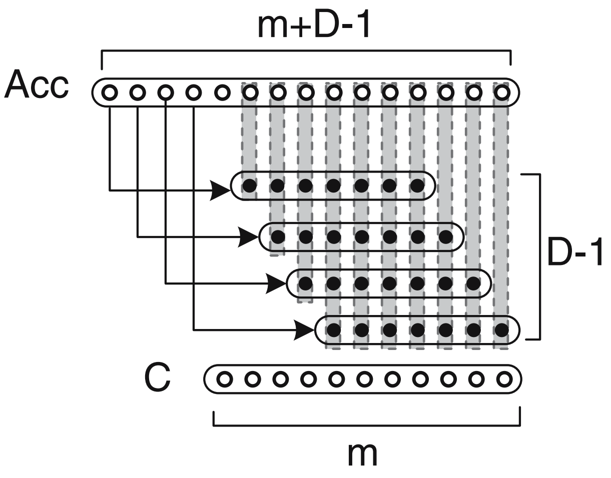


Figure 12 SAM main reduction circuit.



We provide here an analysis of the area requirements and the critical path of the different components of the multiplier. In what follows, we will refer to multipliers with a single accumulator as Single-Accumulator-Multipliers or SAM for short. In Figures 11, 12, and 13, we denote an AND gate with a filled dot and elements to be XORed by a shaded line over them. The number of XOR gates and the critical path is based on the assumption that a binary tree structure is used to XOR the required elements. For n elements, the number of XOR gates required is $n-1$ and the critical path delay becomes the binary tree depth $\lceil \log_2 n \rceil$. We calculate the critical path as a function of the delay of one XOR gate(Δ_{XOR}) and one AND gate (Δ_{AND}). This allows our analysis to be independent of the cell-technology used for the implementation.

Figure 13 SAM final reduction circuit.



4.1.4 SAM Core

The multiplier core performs the operation $C \leftarrow B_i A + C$ (Step 4 Algorithm 5). The implementation of the multiplier core is as shown in Figure 11 for a digit size $D = 4$. It consists of ANDing the multiplicand A with each element of the digit of the multiplier B and XORing the result into the accumulator Acc. The multiplier core requires mD AND gates (denoted by the black dots), mD XOR gates (for XORing the columns denoted by the shaded line) and $m + D - 1$ Flip-Flops (FF) for accumulating the result C.

It can be seen that the multiplier core has a maximum critical path delay of one Δ_{AND} (since all the ANDings in one column are done in parallel) and the delay for XORing $D + 1$ elements as shown in Figure 11. Thus the total critical path delay of the multiplier core is $\Delta_{AND} + \lceil log_2(D + 1)\rceil \Delta_{XOR}$.

4.1.5 SAM Main Reduction Circuit

The main reduction circuit performs the operation $A \leftarrow A\alpha^D \bmod F(\alpha)$ (Step 3 Algorithm 5) and is implemented as in Figure 12. Here the multiplicand A is shifted left by the digit-size D which is equivalent to multiplying by α^D. The result is then reduced with the reduction polynomial by ANDing the higher D elements of the shifted multiplicand with the reduction polynomial $F(\alpha)$ (shown in the figure as pointed arrows) and XORing the result. We assume that the reduction polynomial is chosen according to Theorem 2 which allows reduction to be done in one single step. We can then show that the critical path delay of the reduction circuit is equal to or less than that of the multiplier core.

The main reduction circuit requires $(k + 1)$ ANDs and k XORs gates for each reduction element. The number of XOR gates is one less because the last element of the reduction are XORed to empty elements in the shifted A. Therefore a total of $(k + 1)D$ AND and kD XOR are needed for D digits. Further m FF are needed to store A and $k + 1$ FFs to store the general reduction polynomial.

The critical path of the main reduction circuit (as shown in Figure 12) is one AND (since the ANDings occur in parallel) and the critical path for summation of the D reduction components with the original shifted A. Thus the maximum critical path delay is $\Delta_{AND} + \lceil log_2(D + 1)\rceil \Delta_{XOR}$, which is the same as the critical path delay of the multiplier core.

4.1.6 SAM Final Reduction Circuit

The final reduction circuit performs the operation $C \bmod F(\alpha)$, where C of size $m + D - 2$. It is implemented as shown in Figure 13 which is similar to the main reduction circuit without any shifting. Here the most significant $(D - 1)$ elements are reduced using the reduction polynomial $F(\alpha)$ similarly shown with arrows. The area requirement for this circuit is $(k + 1)(D - 1)$ AND gates and $(k + 1)(D - 1)$ XOR gates. The critical path of the final reduction circuit is $\Delta_{AND} + \lceil log_2(D)\rceil \Delta_{XOR}$ which is less than that of the main reduction circuit since the size of the polynomial reduced is one smaller (Figure 13).

An r-nomial reduction polynomial satisfying Theorem 2, i.e. $\sum_{i=0}^{k} g_i = (r - 1)$, is a special case and hence the critical path is upper-bounded by that obtained for the general case. For a fixed r-nomial reduction polynomial, the area for the main reduction circuit is $(r-1)D$ ANDs, $(r-2)D$ XORs and m FFs. No flip flops are required to store the reduction polynomial as it can be hardwired.

4.2 Squaring in $\mathbb{F}_{2^m}$

Polynomial basis squaring of $C \in \mathbb{F}_{2^m}$ is implemented by expanding C to double its bit-length by interleaving 0 bits in between the original bits of C and then reducing the double length result as shown here:

$$\begin{aligned} C &\equiv A^2 \bmod F(\alpha) \\ &\equiv (a_{m-1}\alpha^{2(m-1)} + a_{m-2}\alpha^{2(m-2)} + \ldots + a_1\alpha^2 + a_0) \bmod F(\alpha) \end{aligned}$$

In hardware, these two steps can be combined if the reduction polynomial has a small number of non-zero coefficients such as in the case of irreducible trinomials and pentanomials. The architecture of the squarer implemented as a hardwired XOR circuit is shown in Figure 14. Here, the squaring is efficiently implemented for $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$, to generate the result in *one single clock cycle* without huge area requirements. It involves first the expansion by interleaving with 0s, which in hardware is just an interleaving of 0 bit valued lines on to the bus to expand it to $2n$ bits. The reduction of this polynomial is inexpensive, first, due to the fact that reduction polynomial used is a pentanomial, and secondly, the polynomial being reduced is sparse with no reduction required for $\lfloor n/2 \rfloor$ of the higher order bits (since they have been set to 0s).

The XOR requirements and the maximum critical path (assuming an XOR tree implementation) for three different reduction polynomials used in elliptic curve cryptography are given in Table 3.

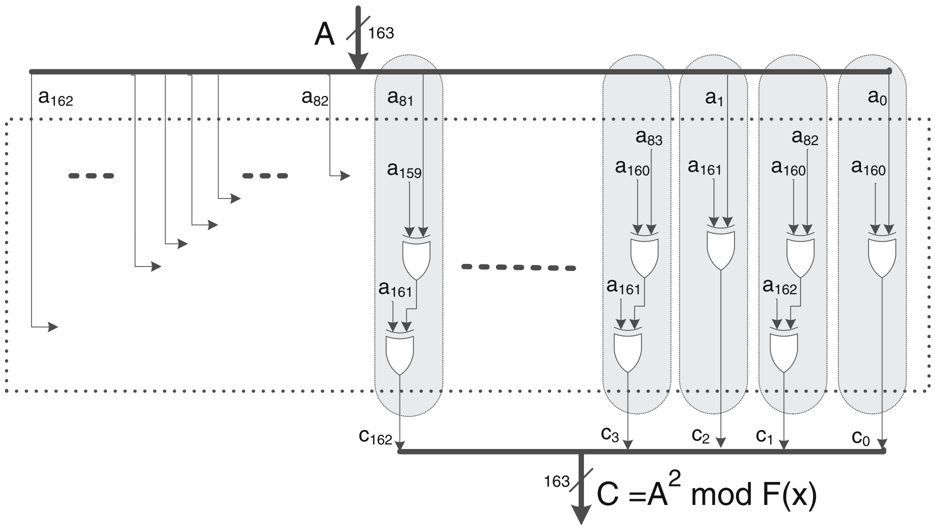


Figure 14 $\mathbb{F}_{2^{163}}$ squaring circuit.

Table 3 $\mathbb{F}_{2^m}$ squaring unit requirements

Reduction polynomial F(x)	XOR gates	Critical path
$x^{131}+x^8+x^3+x^2+1$	205 XOR	$3\ \Delta_{XOR}$
$x^{163}+x^7+x^6+x^3+1$	246 XOR	$3\ \Delta_{XOR}$
$x^{193}+x^15+1$	96 XOR	$2\ \Delta_{XOR}$

5 Hardware Implementation Techniques for Fields $\mathbb{F}_{p^m}$

In recent years, there has been increased interest in cryptographic systems based on fields of odd characteristic [13, 14, 41, 45, 46, 58, 64]. This section is concerned with hardware architectures for addition, multiplication, and inversion in $\mathbb{F}_{p^m}$. The multiplier architectures that we describe are completely general [6, 8] in the sense that they can be applied to any extension degree m. We also study carefully the case of $\mathbb{F}_{3^m}$ due to its cryptographic significance as applied in identity-based cryptosystems and short signature schemes. Finally, we describe exponentiation based techniques for inversion based on the treatment of [27].

5.1 Adder Architectures for $\mathbb{F}_{p^m}$

Addition in $\mathbb{F}_{p^m}$ is performed according to Eq. (16). A parallel adder requires m $\mathbb{F}_p$ adders and its critical path delay is one $\mathbb{F}_p$ adder. In some multiplier architectures, such as the Most Significant Digit-Element (MSDE) first multiplier, the addition of two intermediate polynomials of degree larger than m might need to be performed. In these cases, a parallel adder will require $(m+D)$ $\mathbb{F}_p$ adders but the critical path delay will remain that of one $\mathbb{F}_p$ adder.

5.2 Serial Architectures: LSE and MSE Multipliers over $\mathbb{F}_{p^m}$

There are three different types of architectures used to build $\mathbb{F}_{p^m}$ multipliers: array-, digit-, and parallel-multipliers [65]. Array-type (or serial) multipliers process all coefficients of the multiplicand in parallel in the first step, while the coefficients of the multiplier are processed serially. Array-type multiplication can be performed in two different ways, depending on the order in which the coefficients of the multiplier are processed: Least Significant Element (LSE) first multiplier and Most Significant Element (MSE) first multiplier.

5.2.1 Least Significant Element (LSE) First Multiplier

As in the $\mathbb{F}_{2^m}$ case, the LSE scheme over $\mathbb{F}_{p^m}$ processes first coefficient b_0 of the multiplier and continues with the remaining coefficients one at the time in ascending order. Hence, multiplication according to this scheme can be performed in the following way:

$$\begin{aligned} C &\equiv AB \bmod F(\alpha) \\ &\equiv b_0A + b_1(A\alpha \bmod F(\alpha)) + b_2(A\alpha^2 \bmod F(\alpha)) + \ldots + b_{m-1}(A\alpha^{m-1} \bmod F(\alpha)) \end{aligned}$$

Algorithm 6 LSE Multiplier

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$, where $a_i, b_i \in \mathbb{F}_p$
Output: $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in \mathbb{F}_p$
1: $C \leftarrow 0$
2: for $i = 0$ to $m - 1$ do
3:     $C \leftarrow b_i A + C$
4:     $A \leftarrow A\alpha \bmod F(\alpha)$
5: end for
6: Return (C)
```

The accumulation of the partial product has to be performed with a polynomial adder. This multiplier computes the operation according to Algorithm 6.

5.2.2 Most Significant Element (MSE) First Multiplier

The most significant element multiplication starts with the highest coefficient of the multiplier polynomial. Hence, the multiplication can be performed in the following way:

$$\begin{aligned} C &\equiv AB \bmod F(\alpha) \\ &\equiv (\ldots(b_{m-1}A\alpha \bmod F(\alpha) + b_{m-2}A)\alpha \bmod F(\alpha) + \ldots + b_1 A)\alpha \bmod F(\alpha) + b_0 A \end{aligned}$$

The algorithm is similar to the characteristic two case except that instead of bits we process elements of $\mathbb{F}_p$, i.e., $\lceil \log_2(p) \rceil$ bits. Similarly whenever in the binary case we shift by one bit (multiplication by α), in the odd characteristic case, we need to shift by $\lceil \log_2(p) \rceil$ bits.

5.2.3 Reduction mod $F(\alpha)$

In both LSE and MSE multipliers a quantity $W\alpha$, where $W = \sum_{i=0}^{m-1} w_i\alpha^i \in \mathbb{F}_{p^m}$, $w_i \in \mathbb{F}_p$, has to be reduced mod $F(\alpha)$. It can be shown that $W\alpha \bmod F(\alpha)$ can then be calculated as follows:

$$W\alpha \bmod F(\alpha) = \sum_{i=0}^{m-1}(-g_i w_{m-1})\alpha^i + \sum_{i=1}^{m-1} w_i\alpha^i = (-g_0 w_{m-1}) + \sum_{i=1}^{m-1}(w_{i-1} - g_i w_{m-1})\alpha^i$$

where all coefficient arithmetic is done modulo p. Once again we emphasize that the only differences between odd characteristic and even characteristic multipliers is that in the first case the coefficients are elements of $\mathbb{F}_p$ and that implies that the coefficients are groups of $\lceil \log_2(p) \rceil$ bits. In addition, the reduction circuitry requires adders and subtracters as opposed to only adders (XOR gates) as in the characteristic two case.

5.3 Digit-Serial/Parallel Multipliers for $\mathbb{F}_{p^m}$

As in the characteristic 2 case, we can process more than coefficient of the multiplicand at the time. This results in digit multipliers. The number of coefficients that are processed in parallel is defined to be the digit-size and we denote it with the letter D. For a digit-size D, we can denote by $d = \lceil m/D \rceil$ the total number of digits in a polynomial of degree $m-1$. Digit multipliers for $\mathbb{F}_{p^m}$ are similar to their binary characteristic counterparts except that instead of groups of bits now we need to process groups of coefficients in parallel. As in [65], we can re-write the multiplier as $B = \sum_{i=0}^{d-1} B_i \alpha^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j \;\; 0 \le i \le d-1$ and we assume that B has been padded with zero coefficients such that $b_i = 0$ for $m-1 < i < d \cdot D$ (i.e. the size of B is $d \cdot D$ coefficients but $\deg(B) < m$). Hence,

$$C \equiv AB \bmod F(\alpha) = A \sum_{i=0}^{d-1} B_i \alpha^{Di} \bmod F(\alpha)$$

As in the binary case, depending on the way we process the digits of the polynomial B, the multipliers can be classified as Least Significant Digit-Element first multiplier (LSDE) and Most Significant Digit-Element first multiplier (MSDE). Here, we have introduced the word *element* to clarify that the digits correspond to groups of $\mathbb{F}_p$ coefficients in contrast to [65] where the digits were groups of bits. The algorithms themselves are simple generalizations of the $\mathbb{F}_{2^m}$ case and so we refer the reader to Section 4 or to [6, 30] where the treatment is explicit for $\mathbb{F}_{p^m}$. Finally, notice that all ground field arithmetic is performed in $\mathbb{F}_p$. Thus, $\mathbb{F}_p$ multipliers and adders/subtracters are required to build $\mathbb{F}_{p^m}$ multipliers in contrast to the $\mathbb{F}_{2^k}$ case, where only AND and XOR gates are required. Subtracters (multiplication by -1) can be implemented as multiplication by $p-1$. However, multiplication by α^D is very similar in both $\mathbb{F}_{2^k}$ and $\mathbb{F}_{p^m}$ fields. The only difference is that instead of shifting D bits (as in the $\mathbb{F}_{2^k}$ case), one has to shift $D\lceil \log_2(p) \rceil$ bits in $\mathbb{F}_{p^m}$ fields.

5.4 Systolic and Scalable Architectures for Digit-Serial Multiplication

The work in [6] describes architectures for digital multiplication in $\mathbb{F}_{p^m}$ but their methods have the drawback of using global signals and long wires and they require reconfigurability to achieve their full potential. In particular, [6] uses irreducible trinomial specific circuitry to perform modular reduction on FPGAs. Thus, these solutions lack flexibility in other hardware platforms such as ASICs. In this section we describe the systolic architectures introduced in [8] for $\mathbb{F}_{p^m}$ fields. The work in [8] has several advantages over standard digit multipliers, which include:

- By using a systolic design we use localized routing, thus avoiding the need for global control signals and long wires.
- Their methodology allows for ease of design and offers functional and layout modularity all of which are properties envisioned in good VLSI designs
- The authors in [8] modify the standard digit multiplier designs to allow for scalability as introduced in [70]. In other words, for a fixed value of the digit-size D [6, 65] and parameter d, we can perform a multiplication for any value of m in $\mathbb{F}_{p^m}$, with fixed p, i.e., multiple irreducible polynomials are supported, making unnecessary the use of reconfigurability in FPGAs.

5.4.1 Systolic Least-Significant Digit Element (LSDE) First Architecture

The basic idea in [8] is to make modular reduction independent of the irreducible modulus $F(\alpha)$, by defining an alternate modulus $\overline{F}(\alpha)$, working modulo $\overline{F}(\alpha)$ during the exponentiation (scalar multiplication if considering elliptic curves) phase of the cryptographic operation and then at the end reducing modulo $F(\alpha)$ to obtain the final result. Before continuing, we illustrate the problem that [8] is trying to solve with an example.

Example 1 Suppose you want to compute $A\alpha^D \bmod F(\alpha)$ where $A \in \mathbb{F}_{p^m}$ and $F(\alpha) = \alpha^m + g_k\alpha^k + \sum_{i=0}^{k-1} g_i\alpha^i$ is an optimum irreducible polynomial in the sense of [6, 65]. Let $A = \sum_{i=0}^{d-1} A_i\alpha^{Di} \in \mathbb{F}_{p^m}$ with $d = \lceil m/D \rceil$, A_i a digit (i.e., a group of D $\mathbb{F}_p$ coefficients). Then,

$$A\alpha^D \equiv \alpha^D \sum_{i=0}^{d-1} A_i\alpha^{Di} \bmod F(\alpha) = A_{d-1}\alpha^{Dd} + \sum_{i=1}^{d-1} A_{i-1}\alpha^{Di} \bmod F(\alpha)$$

$$A\alpha^D \equiv A_{d-1}\alpha^{Dd-m}\left(-g_k\alpha^k - \sum_{i=0}^{k-1} g_i\alpha^i\right) + \sum_{i=1}^{d-1} A_{i-1}\alpha^{Di}$$

Notice that the first term in the reduced result depends on the value of m, in other words on the field size. In fact, one needs to multiply by α^{Dd-m}, which can be instantiated as a *variable* shifter in hardware. This is undesirable if scalability of the multiplier is desired.

The following proposition is the basis for the architecture presented in [8].

Proposition 1 [8] Let $A, B \in \mathbb{F}_{p^m}$, $F(\alpha) = \alpha^m + \sum_{i=0}^{m-1} g_i\alpha^i$, be an irreducible polynomial over $\mathbb{F}_p$, and $d = \lceil m/D \rceil$. Then, $A \cdot B \bmod F(\alpha) \equiv [A \cdot B \bmod \overline{F}(\alpha)] \bmod F(\alpha)$, where $\overline{F}(\alpha) = \alpha^{Dd-m}F(\alpha)$.

Intuitively, Proposition 1 says that we can perform reductions modulo $\overline{F}(\alpha) = \alpha^{Dd-m}F(\alpha)$ and still obtain a result which when reduced modulo $F(\alpha)$ returns the correct value. Algorithm 7 shows an LSDE multiplier incorporating the modified modulus of Proposition 1.

Algorithm 7 suggests the following computation strategy. Given two inputs A, $B \in \mathbb{F}_{p^m}$ one can compute $C \equiv A \cdot B \bmod F(\alpha)$ by first computing $\overline{C} \equiv A \cdot B \bmod \overline{F}(\alpha)$ using Algorithm 7 and, then, computing $C \equiv \overline{C} \bmod F(\alpha)$. The second step follows as a consequence of Proposition 1. In practice, the second step can be performed at the end of a long range of computations, similar to the procedure used when performing Montgomery multiplication. Step 4 in Algorithm 7 requires a modular multiplication. Reference [8] defines optimal polynomials which allow to reduce $A\alpha^D$ in just one iteration and make the reduction process independent of the value of m and, thus, of the field $\mathbb{F}_{p^m}$.

Theorem 3 [8] *Let $A = \sum_{i=0}^{d-1} A_i\alpha^{Di}$ be as defined in Algorithm 7 and $\overline{F}(\alpha) = \alpha^{Dd-m}F(\alpha) = \alpha^{Dd} + \sum_{i=0}^{d-1} \overline{F}_i\alpha^{Di}$ be such that $F(\alpha)$ is irreducible over $\mathbb{F}_p$ of degree*

Algorithm 7 Modified LSDE Multiplier

```
Input: $A = \sum_{i=0}^{d-1} A_i\alpha^{Di}$ with $A_i = \sum_{j=0}^{D-1} a_{Di+j}\alpha^j$, $B = \sum_{i=0}^{d-1} B_i\alpha^{Di}$ with $B_i = \sum_{j=0}^{D-1} b_{Di+j}\alpha^j$, $\overline{F}(\alpha) = \alpha^{Dd-m}F(\alpha)$, $a_i, b_i \in \mathbb{F}_p$, and $d = \lceil \frac{m}{D} \rceil$
Output: $\overline{C} \equiv A \cdot B \bmod \overline{F}(\alpha) = \sum_{i=0}^{d} \overline{C}_i\alpha^{Di}$ with $\overline{C}_i = \sum_{j=0}^{D-1} c_{Di+j}\alpha^j$, $c_i \in \mathbb{F}_p$, and $d = \lceil \frac{m}{D} \rceil$
1: $\overline{C} \leftarrow 0$
2: for $i = 0$ to $d-1$ do
3:     $\overline{C} \leftarrow B_iA + \overline{C}$
4:     $A \leftarrow A\alpha^D \bmod \overline{F}(\alpha)$
5: end for
6: Return $(\overline{C} \bmod \overline{F}(\alpha))$
```

m. Then, if $\overline{F}_{d-1} = 0$ *or* $\overline{F}_{d-1} = 1$, $A\alpha^D \bmod \overline{F}(\alpha)$ *can be computed in one reduction step. Moreover,* $\overline{F}_{d-1} = 0$ *implies that for* $F(\alpha) = \alpha^m + \sum_{i=0}^{m-1} g_i\alpha^i$, *coefficients* $g_{m-1} = g_{m-2} = \cdots = g_{m-D} = 0$. *Similarly, when* $\overline{F}_{d-1} = 1$ *then* $g_{m-1} = g_{m-2} = \cdots = g_{m-D+1} = 0$.

Notice that Theorem 3 implies that if $F(\alpha) = \alpha^m + g_k\alpha^k + \sum_{i=0}^{k-1} g_i\alpha^i$ is to be an optimal M-LSDE polynomial, then $k \le m - D$. This agrees with the findings in [6, 65]. Notice also that the way modular reduction is performed in Step 4 of Algorithm 7 is independent of the value of m and thus of the field. The price of this field independence is that now we do not obtain anymore the value of $A \cdot B \bmod F(\alpha)$ but rather $A \cdot B \bmod \overline{F}(\alpha)$ thus, requiring one more reduction at the end of the whole computation. In addition, we need to multiply *once* at initialization $F(\alpha)$ by α^{Dd-m}. This, however, can be thought of as analogous to the Montgomery initialization, and thus, can be neglected when considering the total costs of complex computations which is customary practice in cryptography. In addition, notice that multiplication by α can be easily implemented in hardware via left shifts.

5.5 Comments on Irreducible Polynomials of Degree m over $\mathbb{F}_p$

For fields $\mathbb{F}_{p^m}$ with odd prime characteristic it is often possible to choose irreducible binomials $F(\alpha) = x^m - \omega$, $\omega \in \mathbb{F}_p$. Another interesting property of binomials is that they are optimum from the point of view of Theorem 1. In addition, reduction is virtually for free, corresponding to just a few $\mathbb{F}_p$ multiplications (this follows from the fact that $\alpha^m = \omega$). We notice that the existence of irreducible binomials has been exactly established as Theorem 4 shows.

Theorem 4 [43] *Let* $m \ge 2$ *be an integer and* $\omega \in F_q^{\star}$. *Then the binomial* $x^m - \omega$ *is irreducible in* $F_q[x]$ *if and only if the following two conditions are satisfied: (a) each prime factor of* m *divides the order* e *of* ω *in* $F_q^{\star}$, *but not* $(q-1)/e$; *(b)* $q \equiv 1 \bmod 4$ *if* $m \equiv 0 \bmod 4$.

When irreducible binomials can not be found, one searches in incremental order for irreducible trinomials, quadrinomials, etc. In von zur Gathen and Nöcker [73] conjecture that the minimal number of terms $\sigma_q(m)$ in irreducible polynomials of degree m over $\mathbb{F}_q$, q a power of a prime, is for $m \geq 1$, $\sigma_2(m) \leq 5$ and $\sigma_q(m) \leq 4$ for $q \geq 3$. This conjecture has been verified for $q = 2$ and $m \leq 10,000$ [7, 25, 73, 78–80] and for $q = 3$ and $m \leq 539$ [72].

By choosing irreducible polynomials with the least number of non-zero coefficients, one can reduce the area complexity of the LSDE multiplier. Reference [30] notices that by choosing irreducible polynomials such that their non-zero coefficients are all equal to $p-1$ one can further reduce the complexity of the multiplier. Notice that there is no existence criteria for irreducibility of trinomials over any field $\mathbb{F}_{p^m}$. The most recent advances in this area are the results of Loidreau [44], where a table that characterizes the parity of the number of factors in the factorization of a trinomial over $\mathbb{F}_3$ is given, and the necessary (but not sufficient) irreducibility criteria for trinomials introduced by von zur Gathen in [72]. Reference [6] provides tables of irreducible polynomials over $\mathbb{F}_3$ for degrees $1 < m < 256$.

5.6 Case Study: $\mathbb{F}_{3^m}$ Arithmetic

To our knowledge, [59] is the first to describe $\mathbb{F}_{3^m}$ architectures for applications of cryptographic significance. The authors introduce a representation similar to the one used by [24] to represent their polynomials. In particular, they combine all the least significant bits of the coefficients of an element, say A, into one value and all the most significant bits of the coefficients of A into a second value (notice the coefficients of A are elements of $\mathbb{F}_3$ and thus 2 bits are needed to represent each of them). Thus, $A = (a_1, a_0)$ where a_1 and a_0 are m-bit long each. Addition of two polynomials $A = (a_1, a_0)$, $B = (b_1, b_0)$ with $C = (c_1, c_0) \equiv A + B$ is achieved as:

$$\begin{aligned} t &= (a_1 \vee b_0) \oplus (a_0 \vee b_1) \\ c_1 &= (a_0 \vee b_0) \oplus t \\ c_0 &= (a_1 \vee b_1) \oplus t \end{aligned} \tag{21}$$

where $\vee$ and $\oplus$ mean the logical OR and exclusive OR operations, respectively. Page and Smart [59] notice that subtraction and multiplication by 2 are equivalent in characteristic 3 and that they can be achieved as $2 \cdot A = 2 \cdot (a_1, a_0) = -A = -(a_1, a_0) = (a_0, a_1)$. Multiplication is achieved in the bit-serial manner, by repeatedly shifting the multiplier down by one bit position and shifting the multiplicand up by one bit position. The multiplicand is then added or subtracted depending on whether the least significant bit of the first or second word of the multiplier is equal to 1. The authors do not mention what methods were used to perform modular reduction in the field. Reference [59] also notices that with this representation a cubing operation can only be as fast as a general multiply, whereas, using other implementation methods the cubing operation could be much faster. The implementation of multiplication in $\mathbb{F}_{(3^m)^6}$ is also discussed using the irreducible polynomial $Q(y) = y^6 + y + 2$. They use the normal method to multiply polynomials of degree 5 with coefficients in $\mathbb{F}_{3^m}$ and then reduce modulo $Q(y)$ using 10 additions and 4 doublings in $\mathbb{F}_{3^m}$. In addition, they suggest that using the Karatsuba algorithm for multiplication [38], performance can be improved at the cost of additional area.

5.6.1 $\mathbb{F}_3$ *Arithmetic Implementation on FPGAs*

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices whose basic logic elements are Look-Up Tables (LUTs), sometimes also called Configurable Logic Blocks (CLBs), Flip-Flops (FFs), and, for modern devices, memory elements [1, 2, 77]. The LUTs are used to implement Boolean functions of their inputs, that is, they are used to implement functions traditionally implemented with logic gates. In the particular case of the XCV1000E-8-FG1156 and the XC2VP20-7-FF1156, their basic building blocks are 4-bit input/1-bit output LUTs. This means that all basic arithmetic operations in $\mathbb{F}_3$ (add, subtract, and multiply) can be done with two LUTs, where each LUT generates 1 bit of the output. This follows from the fact that any of these arithmetic operations over $\mathbb{F}_3$ can be thought of as logic functions in four input variables a_1, a_0, b_1, b_0 and two output variables c_1, c_0 as:

$$f: \quad I^4 \longrightarrow O^2$$

where $I = \{0, 1\}$ and $O = \{0, 1\}$. Then, given three elements $a = (a_1, a_0)_2, b = (b_1, b_0)_2, c = (c_1, c_0)_2 \in \mathbb{F}_3$, we can write the function 'multiplication in $\mathbb{F}_3$' as Table 4.

In Table 4, we have assumed that (1, 1) is an alternate representation for $0 \in \mathbb{F}_3$. Notice that it is possible to choose different representations as shown in [31]. This might minimize the complexity of the $\mathbb{F}_3$ multiplier in ASIC-based designs. However, in FPGA based designs, a different encoding has no advantages because of the LUT-based structure of the FPGA.

5.6.2 *Cubing in* $\mathbb{F}_{3^m}$

It is well known that for $A \in \mathbb{F}_{p^m}$ the computation of A^p (also known as the Frobenius map) is linear. In the particular case of $p = 3$, we can write the Frobenius map as:

$$A^3 \bmod F(\alpha) = \left(\sum_{i=0}^{m-1} a_i \alpha^i \right)^3 \bmod F(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^{3i} \bmod F(\alpha) =$$

Table 4 Truth table representing multiplication in $\mathbb{F}_3$

$a_1\, a_0\, b_1\, b_0$	$c_1\, c_0$	$a_1\, a_0\, b_1\, b_0$	$c_1\, c_0$
0000	00	1000	00
0001	00	1001	10
0010	00	1010	01
0011	00	1011	00
0100	00	1100	00
0101	01	1101	00
0110	10	1110	00
0111	00	1111	00

which can in turn be written as the sum of three terms (notice that here we have re-written the indexes in the summation):

$$A^3 \bmod F(\alpha) = \sum_{\substack{i=0 \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}} \alpha^i \bmod F(\alpha) = T + U + V \bmod F(\alpha) =$$

$$= \left(\sum_{\substack{i=0 \\ i\equiv 0 \bmod 3}}^{m-1} a_{\frac{i}{3}} \alpha^i \right) + \left(\sum_{\substack{i=m \\ i\equiv 0 \bmod 3}}^{2m-1} a_{\frac{i}{3}} \alpha^i \right) + \left(\sum_{\substack{i=2m \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}} \alpha^i \right) \bmod F(\alpha)$$

Notice that only U and V need to be reduce mod $F(\alpha)$. Reference [6] further assumes that $F(x) = x^m + g_t x^t + g_0$ and that $t < m/3$, which proves to be a valid assumption in terms of the existence of such irreducible trinomials. Then, it can be shown that:

$$U = \sum_{\substack{i=m \\ i\equiv 0 \bmod 3}}^{2m-1} a_{\frac{i}{3}} \alpha^i \bmod F(\alpha) = \sum_{\substack{i=m \\ i\equiv 0 \bmod 3}}^{2m-1} a_{\frac{i}{3}} \alpha^{i-m} \left(-g_t \alpha^t - g_0\right) \bmod F(\alpha)$$

$$V = \sum_{\substack{i=2m \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}} \alpha^i \bmod F(\alpha) = \sum_{\substack{i=2m \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}} \alpha^{i-2m} \left(\alpha^{2t} - g_t g_0 \alpha^t + 1\right) \bmod F(\alpha)$$

It can also be shown that U and V can be reduced to be of degree less than m in one extra reduction step. Assuming that $F(\alpha)$ is a trinomial with $t < m/3$ and that the circuit is implemented for fixed irreducible trinomials, the cubing circuit can be implemented in about $2m$ adders/subtracters.

5.7 Non-general Multipliers

In contrast to the $\mathbb{F}_p$ case, there has not been a lot of work done on $\mathbb{F}_{p^m}$ architectures. Our literature search yielded [56] and more recently [9, 28] as the only references that explicitly treated the general case of $\mathbb{F}_{p^m}$ multipliers, p an odd prime. Reference [54] treats explicitly the case of $\mathbb{F}_{(3^n)^3}$. We do not discuss [54] and [76], who introduced parallel multipliers for $\mathbb{F}_{p^m}$, as parallel multipliers are not well suited to cryptographic applications due to their excessive hardware requirements.

In [56], $\mathbb{F}_{p^m}$ multiplication is computed in two stages:

1. The polynomial product is computed modulo a highly factorisable degree S polynomial, $M(x)$, with $S \geq 2m - 1$. This restriction comes from the fact that the product of two polynomials of maximum degree $m - 1$ is at most $2m - 1$. Then, the product is computed using a polynomial residue number system (PRNS), originally introduced in [68]. This involves converting back and forth between the normal representation and the PRNS representation.
2. The second step involves reducing modulo the irreducible polynomial $q(x)$ over which $\mathbb{F}_{p^m}$ is defined.

In order to further simplify the complexity of these multipliers, [56] suggests to limit the form of $M(x)$ to being fully factorisable into degree-1 polynomials. Parker and Benaissa [56] show that this is equivalent to requiring that the inequality $2m < p$ be satisfied. This restriction implies that for all primes $p < 67$, these multipliers can not be implemented if intended for use in cryptographic applications.[3] A second optimization in [56] is to consider only fields $\mathbb{F}_{p^m}$ for which an irreducible binomial of degree m over $\mathbb{F}_p$ exists. It turns out that the second optimization reduces significantly the number of fields for which these multipliers are of interest. We notice that the number of fields for which these multipliers are feasible might be increased by considering higher-dimensional PRNS as suggested in [56]. However, this technique requires that m be a composite integer which in many cryptographic applications is seen with skepticism because of security considerations. The architectures presented in [9] are similarly constrained, i.e., they can only be implemented for $p \geq 67$ if the desired group size is 2^{160}, however, there are no restrictions similar to [56].

5.8 Parallel Multipliers for $\mathbb{F}_{p^m}$

In general, parallel multipliers require too many hardware resources to be implemented in a realistic environment for the field sizes required in cryptography. However, for other applications they might prove to be the right choice. Thus, we provide references to some work in this area. Reference [76] introduces a parallel multiplier in $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ using a weakly dual basis. The multiplier has complexity of at most m^2 multipliers and $(k-1)(m-1)$ constant multipliers in $\mathbb{F}_q$, $m^2 + (k-3)m - (k-2)$ adders (or subtracters) and $(m-1)$ constant adders (or subtracters) in $\mathbb{F}_q$, where the irreducible polynomial has k non-zero coefficients.

The authors in [54] consider multiplier architectures for composite fields of the form $\mathbb{F}_{(3^n)^3}$ using Multi-Value Logic (MVL) and a modified version of the Karatsuba algorithm [38] for polynomial multiplication over $\mathbb{F}_{(3^n)^3}$. Elements of $\mathbb{F}_{(3^n)^3}$ are represented as polynomials of maximum degree 2 with coefficients in $\mathbb{F}_{3^n}$. Multiplication in $\mathbb{F}_{3^n}$ is achieved in the obvious way. Karatsuba multiplication is combined with modular reduction over $\mathbb{F}_{(3^n)^m}$. to reduce the complexity of their design. Because of the use of MVL no discussion of modulo 3 arithmetic is given. The authors estimate the complexity of their design for arithmetic over $\mathbb{F}_{(3^2)^3}$ as 56 mod-3 adders and 67 mod-3 multipliers.

6 Itoh–Tsujii Inversion in Fields $\mathbb{F}_{p^m}$

Originally introduced in [32], the Itoh and Tsujii algorithm (ITA) is an exponentiation-based algorithm for inversion in finite fields which reduces the complexity of computing the inverse of a non-zero element in $\mathbb{F}_{2^k}$, when using a normal basis representation, from $n-2$ multiplications in $\mathbb{F}_{2^k}$ and $n-1$ cyclic shifts using the binary exponentiation method to at most $2\lfloor \log_2(n-1) \rfloor$ multiplications in

[3] It is widely accepted that for cryptosystems against which the Pollard's rho algorithm or one of its variants are the best available attacks, such as elliptic curve cryptosystems, the group order should be greater or equal to 2^{160}. Thus, solving $2m < p$ and $p^m \geq 2^{160}$ for p and m, one obtains $p \geq 67$. Notice that the value of p grows as the size of the desired group grows. For groups with $|G| \geq 2^{192}$, $|G| \geq 2^{223}$, and $|G| \geq 521$, the prime p satisfies $p \geq 67$, $p \geq 79$, and $p \geq 157$, respectively.

$\mathbb{F}_{2^k}$ and $n-1$ cyclic shifts. As shown in [27], the method is also applicable to finite fields with a polynomial basis representation.

Now, we can show how to compute the multiplicative inverse of $A \in \mathbb{F}_{2^k}$, $A \neq 0$, according to the binary method for exponentiation. From Fermat's little theorem we know that $A^{-1} \equiv A^{2^n-2}$ can be computed as

$$A^{2^n-2} = A^2 \cdot A^{2^2} \cdots A^{2^{n-1}}$$

This requires $n-2$ multiplications and $n-1$ cyclic shifts. Notice that because we are working over a field of characteristic 2, squaring is a linear operation. In addition, if a normal basis is being used to represent the elements of the field, we can compute A^2 for any $A \in \mathbb{F}_{2^k}$ with one cyclic shift.

Itoh and Tsujii proposed in [32] three algorithms. The first two algorithms describe addition chains for exponentiation-based inversion in fields $\mathbb{F}_{2^k}$ while the third one describes a method based on subfield inversion. The first algorithm is only applicable to values of n such that $n = 2^r + 1$, for some positive r, and it is based on the observation that the exponent $2^n - 2$ can be re-written as $(2^{n-1} - 1) \cdot 2$. Thus if $n = 2^r + 1$, we can compute $A^{-1} \equiv (A^{2^{2^r}-1})^2$. Furthermore, we can re-write $2^{2^r} - 1$ as

$$2^{2^r} - 1 = \left(2^{2^{r-1}} - 1\right) 2^{2^{r-1}} + \left(2^{2^{r-1}} - 1\right) \tag{22}$$

Equation (22) and the previous discussion lead to Algorithm 8.

Notice that Algorithm 8 performs $r = \log_2(n-1)$ iterations. In every iteration, one multiplication and i cyclic shifts, for $0 \leq i < r$, are performed which leads to an overall complexity of $\log_2(n-1)$ multiplications and $n-1$ cyclic shifts.

Algorithm 8 Multiplicative Inverse Computation in $\mathbb{F}_{2^k}$ with $n = 2^r + 1$ [32, Theorem 1]

```
Input: $A \in \mathbb{F}_{2^k}$, $A \neq 0$, $n = 2^r + 1$
Output: $C = A^{-1}$
  $C \leftarrow A$
  for $i = 0$ to $r - 1$ do
    $D \leftarrow C^{2^{2^i}}$ {NOTE: $2^i$ cyclic shifts}
    $C \leftarrow C \cdot D$
  end for
  $C \leftarrow C^2$
  Return (C)
```

Example 2 Let $A \in \mathbb{F}_{2^{17}}$, $A \neq 0$. Then according to Algorithm 8 we can compute A^{-1} with the following addition chain:

$$\begin{aligned} A^2 \cdot A &= A^3 \\ \left(A^3\right)^{2^{2^1}} \cdot A^3 &= A^{15} \\ \left(A^{15}\right)^{2^{2^2}} \cdot A^{15} &= A^{255} \\ \left(A^{255}\right)^{2^{2^3}} \cdot A^{255} &= A^{65535} \\ \left(A^{65535}\right)^2 &= A^{131070} \end{aligned}$$

A quick calculation verifies that $2^{17} - 2 = 131,070$. Notice that in accordance with Algorithm 8 we have performed four multiplications in $\mathbb{F}_{2^{17}}$ and, if using a normal basis, we would also require $2^4 = 16$ cyclic shifts.

Algorithm 8 can be generalized to any value of n [32]. First, we write $n - 1$ as

$$n - 1 = \sum_{i=1}^{t} 2^{k_i}, \quad \text{where } k_1 > k_2 > \cdots > k_t \tag{23}$$

Using the fact that $A^{-1} \equiv (A^{2^{n-1}-1})^2$ and Eq. (23), it can be shown that the inverse of A can be written as:

$$\left(A^{2^{n-1}-1}\right)^2 = \left[\left(A^{2^{2^{k_t}}-1}\right)\left(\left(A^{2^{2^{k_{t-1}}}-1}\right)\cdots \left[\left(A^{2^{2^{k_2}}-1}\right)\left(A^{2^{2^{k_1}}-1}\right)^{2^{2^{k_2}}}\right]^{2^{2^{k_3}}}\cdots\right)^{2^{2^{k_t}}}\right]^2 \tag{24}$$

An important feature of Eq. (24) is that in computing $A^{2^{2^{k_1}}-1}$ all other quantities of the form $A^{2^{2^{k_i}}-1}$ for $k_i < k_1$ have been computed. Thus, the overall complexity of Eq. (24) can be shown to be:

$$\begin{aligned} \#\text{MUL} &= \lfloor \log_2(n-1) \rfloor + HW(n-1) - 1 \\ \#\text{CSH} &= n - 1 \end{aligned} \tag{25}$$

where $HW(\cdot)$ denotes the Hamming weight of the operand, i.e., the number of ones in the binary representation of the operand, MUL refers to multiplications in $\mathbb{F}_{2^k}$, and CSH refers to cyclic shifts over $\mathbb{F}_2$ when using a normal basis.

Example 3 Let $A \in \mathbb{F}_{2^{23}}$, $A \neq 0$. Then according to Eq. (23) we can write $n-1 = 22 = 2^4 + 2^2 + 2$ where $k_1 = 4$, $k_2 = 2$, and $k_3 = 1$. It follows that we can compute $A^{-1} \equiv A^{2^{23}-2}$ with the following addition chain:

$$\begin{aligned} A^{2^2-1} &= A^2 \cdot A \\ A^{2^4-1} &= \left(A^3\right)^{2^2} \cdot A^3 \\ A^{2^8-1} &= \left(A^{15}\right)^{2^4} \cdot A^{15} \\ A^{2^{16}-1} &= \left(A^{255}\right)^{2^8} \cdot A^{255} \\ A^{2^{23}-2} &= \left(A^{2^2-1} \cdot \left(A^{2^4-1} \cdot \left(A^{2^{16}-1}\right)^{2^4}\right)^{2^2}\right)^2 \end{aligned}$$

The above addition chain requires 6 multiplications and 22 cyclic shifts which agrees with the complexity of Eq. (25).

In [32], the authors also notice that the previous ideas can be applied to extension fields $\mathbb{F}_{q^m}$, $q = 2^n$. Although this inversion method does not perform a complete inversion, it reduces inversion in $\mathbb{F}_{q^m}$ to inversion in $\mathbb{F}_q$. It is assumed that subfield inversion can be done relatively easily, e.g., through table look-up or with the extended Euclidean algorithm. These ideas are summarized in Theorem 5. The presentation here follows [27] and it is slightly more general than [32] as a subfield of the form $\mathbb{F}_{2^n}$ is not required, rather we allow for general subfields $\mathbb{F}_q$.

Theorem 5 [32, Theorem 3] *Let $A \in \mathbb{F}_{q^m}$, $A \neq 0$, and $r = (q^m - 1)/(q-1)$. Then, the multiplicative inverse of an element A can be computed as*

$$A^{-1} = (A^r)^{-1} A^{r-1}. \tag{26}$$

Computing the inverse through Theorem 5 requires four steps:

Step 1 Exponentiation in $\mathbb{F}_{q^m}$, yielding A^{r-1}.
Step 2 Multiplication of A and A^{r-1}, yielding $A^r \in \mathbb{F}_q$.
Step 3 Inversion in $\mathbb{F}_q$, yielding $(A^r)^{-1}$.
Step 4 Multiplication of $(A^r)^{-1} A^{r-1}$.

Steps 2 and 4 are trivial since both A^r, in Step 2, and $(A^r)^{-1}$, in Step 4, are elements of $\mathbb{F}_q$[43]. Both operations can, in most cases, be done with a complexity that is well below that of one single extension field multiplication. The complexity of Step 3, subfield inversion, depends heavily on the subfield $\mathbb{F}_q$. However, in many cryptographic applications the subfield can be small enough to perform inversion very efficiently, for example, through table look-up [19, 26], or by using the Euclidean algorithm. What remains is Step 1, exponentiation to the $(r-1)$th power in the extension field $\mathbb{F}_{q^m}$.

First, we notice that the exponent can be expressed in q-adic representation as

$$r - 1 = q^{m-1} + \cdots + q^2 + q = (1 \cdots 110)_q \tag{27}$$

This exponentiation can be computed through repeated raising of intermediate results to the q-th power and multiplications. The number of multiplications in $\mathbb{F}_{q^m}$

can be minimized by using the addition chain in Eq. (24). Thus, computing A^{r-1} requires [32]:

$$\begin{aligned} \#\mathrm{MUL} &= \lfloor \log_2(m-1) \rfloor + HW(m-1) - 1 \\ \#q\mathrm{-EXP} &= m-1 \end{aligned} \tag{28}$$

where q-EXP refers to the number of exponentiations to the qth power in $\mathbb{F}_q$.

Example 4 Let $A \in \mathbb{F}_{q^{19}}$, $A \neq 0$, $q = p^n$ for some prime p. Then, using the q-adic representation of $r-1$ from Eq. (27) and the addition chain from Eq. (24), we can find an addition chain to compute $A^{r-1} = A^{q^{18}+q^{17}+\cdots+q^2+q}$ as follows. First, we write $m-1 = 18 = 2^4 + 2$ where $k_1 = 4$, and $k_2 = 1$. Then, $A^{r-1} = (A^{q^{16}+q^{15}+\cdots+q^2+q})^{q^2} \cdot (A^{q^2+q})$ and we can compute $A^{q^{16}+q^{15}+\cdots+q^2+q}$ as

$$\begin{aligned} A^{q^2} &= \left(A^q\right)^q \\ A^{q^2+q} &= A^q \cdot A^{q^2} \\ A^{\sum_{i=1}^4 q^i} &= \left(A^{q^2+q}\right)^{q^2} \cdot A^{q^2+q} \\ A^{\sum_{i=1}^8 q^i} &= \left(A^{\sum_{i=1}^4 q^i}\right)^{q^4} \cdot A^{\sum_{i=1}^4 q^i} \\ A^{\sum_{i=1}^{16} q^i} &= \left(A^{\sum_{i=1}^8 q^i}\right)^{q^8} \cdot \left(A^{\sum_{i=1}^8 q^i}\right) \end{aligned}$$

Notice that in computing $A^{q^{16}+q^{15}+\cdots+q^2+q}$, we have computed A^{q^2+q}. The complexity to compute A^{r-1} (and, thus, the complexity to compute A^{-1} if the complexity of multiplication and inversion in $\mathbb{F}_q$ can be neglected) in $\mathbb{F}_{q^{19}}$ is found to be 5 multiplications in $\mathbb{F}_{q^{19}}$ and 18 exponentiations to the qth power in agreement with Eq. (28).

We notice that [32] assumes a normal basis representation of the field elements of $\mathbb{F}_{q^m}$, $q = 2^n$, in which the exponentiations to the qth power are simply cyclic shifts of the m coefficients that represent an individual field element. In polynomial (or standard) basis, however, these exponentiations are, in general, considerably more expensive.

Reference [27] takes advantage of finite field properties and of the algorithm characteristics to improve on the overall complexity of the ITA in polynomial basis. The authors make use of two facts: (a) the algorithm performs alternating multiplications and several exponentiations to the qth power in a row and (b) raising an element $A \in \mathbb{F}_q$, $q = p^n$, to the q^eth power is a linear operation in $\mathbb{F}_{q^m}$, since q is a power of the field characteristic.

In general, computing A^{q^e} has a complexity of $m(m-1)$ multiplications and $m(m-2)+1 = (m-1)^2$ additions in $\mathbb{F}_q$ [27]. This complexity is roughly the same as one $\mathbb{F}_{q^m}$ multiplication, which requires m^2 subfield multiplications if we do not assume fast convolution techniques (e.g., the Karatsuba algorithm [38] for multiplication). However, in polynomial basis representation computing A^{q^e}, where $e > 1$, can be shown to be as costly as a single exponentiation to the qth power. Thus, [27] performs as many subsequent exponentiations to the qth power in one step between multiplications as possible, yielding the same multiplication complexity as in Eq. (28), but a reduced number of q^e-exponentiations. This is summarized in Theorem 6.

Theorem 6 [27, Theorem 2] *Let $A \in \mathbb{F}_{q^m}$. One can compute A^{r-1} where $r-1 = q + q^2 + \cdots + q^{(m-1)}$ with no more than*

$$\#MUL = \lfloor \log_2(m-1) \rfloor + HW(m-1) - 1$$
$$\#q^e\text{-}EXP = \lfloor \log_2(m-1) \rfloor + HW(m-1)$$

operations, where $\#MUL$ and $\#q^e$-EXP refer to multiplications and exponentiations to the q^eth power in $\mathbb{F}_{q^m}$, respectively.

We would like to stress that Theorem 6 is just an upper bound on the complexity of this exponentiation. Thus, it is possible to find addition chains which yield better complexity as shown in [18]. In addition, we see from Theorem 6 that Step 1 of the ITA algorithm requires about as many exponentiations to the q^eth power as multiplications in $\mathbb{F}_{q^m}$ if a polynomial basis representation is being used. In the discussion earlier in this section it was established that raising an element $A \in \mathbb{F}_{q^m}$ to the q^eth power is roughly as costly as performing one multiplication in $\mathbb{F}_{q^m}$. Hence, if it is possible to make exponentiations to the q^eth power more efficient, considerable speed-ups of the algorithm can be expected. Three classes of finite fields are introduced in [27] for which the complexity of these exponentiations is in fact substantially lower than that of a general multiplication in $\mathbb{F}_{q^m}$. These are:

- Fields $\mathbb{F}_{(2^n)^m}$ with binary field polynomials.
- Fields $\mathbb{F}_{q^m}$, $q = p^n$ and p an odd prime, with binomials as field polynomials.
- Fields $\mathbb{F}_{q^m}$, $q = p^n$ and p an odd prime, with binary equally spaced field polynomials (ESP), where a binary ESP is a polynomial of the form $x^{sm} + x^{s(m-1)} + x^{s(m-2)} + \cdots + x^{2s} + x^s + 1$.

7 Summary

We presented here a survey of different finite field architectures that are suitable for hardware implementations of cryptographic systems. The hardware architectures for addition/subtraction, multiplication, and inverse were presented for the three different finite fields popularly used in cryptography: binary fields, prime fields and extension fields. The architectures differ in their area/time complexities and any implementation of a cryptographic system requires proper choice of the appropriate architectures that satisfies the system constraints.

References

1. Actel Corporation: *Actel's ProASIC family, the only ASIC design flow FPGA*. (2001)
2. Altera Corporation: *APEX 20KC programmable logic device data sheet.* (2001)
3. Amanor, D.N., Paar, C., Pelzl, J., Bunimov, V., Schimmler, M.: Efficient hardware architectures for modular multiplication on FPGAs. In: 2005 International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, pp. 539–542. IEEE Circuits and Systems Society, Piscataway, New Jersey, August 2005

4. Barrett, P.: Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology – CRYPTO'86. LNCS, vol. 263, pp. 311–323. Springer, Berlin Heidelberg New York (1987)
5. Boneh, D., Franklin, M.: Identity-Based Encryption from the Weil Pairing. In: Kilian, J. (ed.) Advances in Cryptology – CRYPTO 2001. LNCS, vol. 2139, pp. 213–229. Springer, Berlin Heidelberg New York (2001)
6. Bertoni, G., Guajardo, J., Kumar, S.S., Orlando, G., Paar, C., Wollinger, T.J.: Efficient $GF(p^m)$ arithmetic architectures for cryptographic applications. In: Joye, M. (ed.) Topics in Cryptology – CT-RSA 2003. LNCS, vol. 2612, pp. 158–175. Springer, Berlin Heidelberg New York (2003)
7. Blake, I.F., Gao, S., Lambert, R.J.: Constructive problems for irreducible polynomials over finite fields. In: Gulliver, T.A., Secord, N.P. (eds.) Information Theory and Applications. LNCS, vol. 793, pp. 1–23. Springer, Berlin Heidelberg New York (1993)
8. Bertoni, G., Guajardo, J., Orlando, G.: Systolic and scalable architectures for digit-serial multiplication in fields $GF(p^m)$. In: Johansson, T., Maitra, S. (eds.) Progress in Cryptology – INDOCRYPT 2003. LNCS, vol. 2904, pp. 349–362. Springer, Berlin Heidelberg New York (2003)
9. Bajard, J.-C., Imbert, L., Nègre, C., Plantard, T.: Efficient multiplication in $GF(p^k)$ for elliptic curve cryptography. In: Bajard, J.-C., Schulte, M. (eds.) Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16), pp. 181–187. Santiago de Compostela, Spain, 15–18 June 2003
10. Bucek, J., Lorencz, R.: Comparing subtraction-free and traditional AMI. In: Proceedings of the 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2006), Prague, Czech Republic, 18–21 April 2006. pp. 97–99. IEEE Computer Society, Los Alamitos, CA, USA (2006)
11. Blakley, G.R.: A computer algorithm for calculating the product $A \cdot B$ modulo M. IEEE Trans. Comput. **C-32**(5), 497–500 (1983)
12. Batina, L., Ors, S.B., Preneel, B., Vandewalle, J.: Hardware architectures for public key cryptography. Integration, VLSI J. **34**(6), 1–64 (2003)
13. Bailey, D.V., Paar, C.: Optimal extension fields for fast arithmetic in public-key algorithms. In: Krawczyk, H. (ed.) Advances in Cryptology – CRYPTO '98. LNCS, vol. 1462, pp. 472–485. Springer, Berlin Heidelberg New York (1998)
14. Bailey, D.V., Paar, C.: Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. J. Cryptology **14**(3), 153–176 (2001)
15. Bunimov, V., Schimmler, M.: Area and time efficient modular multiplication of large integers. In: IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands, June 2003
16. Bunimov, V., Schimmler, M., Tolg, B.: A complexity-effective version of montgomery's algorithm. In: Workshop on Complexity Effective Designs, ISCA'02, Anchorage, Alaska, May 2002
17. Di Claudio, E.D., Piazza, F., Orlandi, G.: Fast combinatorial RNS processors for DSP applications. IEEE Trans. Comput. **44**(5), 624–633 (1995)
18. Chung, J.W., Sim, S.G., Lee, P.J.: Fast implementation of elliptic curve defined over $GF(p^m)$ on CalmRISC with MAC2424 coprocessor. In: Koç, Ç.K., Paar, C. (eds.) Workshop on Cryptographic Hardware and Embedded Systems – CHES, 17–18 August 2000. LNCS, vol. 1965, pp. 57–70. Springer, Berlin Heidelberg New York (2000)
19. De Win, E., Bosselaers, A., Vandenberghe, S., De Gersem, P., Vandewalle, J.: A fast software implementation for arithmetic operations in $GF(2^n)$. In: Kim, K., Matsumoto, T. (eds.) Advances in Cryptology – ASIACRYPT '96. Lecture Notes in Computer Science, vol. 1163, pp. 65–76. Springer, Berlin Heidelberg New York (November 1996)
20. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inform. Theory **IT-22**(6), 644–654 (1976)
21. Diffie, W.: Subject: Authenticity of non-secret encryption documents. Available at http://cryptome.org/ukpk-diffie.htm. October 6, 1999 (Email message sent to John Young)
22. Daly, A., Marnane, L., Popovici, E.: Fast modular inversion in the montgomery domain on reconfigurable logic. Technical report, University College Cork, Ireland (2003)
23. Ellis, J.H.: The story of non-secret encryption. Available at http://jya.com/ellisdoc.htm (December 16, 1997)
24. Galbraith, S.D., Harrison, K., Soldera, D.: Implementing the Tate pairing. In: Fieker, C., Kohel, D. (eds.) Algorithmic Number Theory – ANTS-V, LNCS, vol. 2369, pp. 324–337. Springer, Berlin Heidelberg New York (2002)
25. Golomb, S.W.: Shift Register Sequences. Holden-Day, San Francisco, USA (1967)

26. Guajardo, J., Paar, C.: Efficient algorithms for elliptic curve cryptosystems. In: Kaliski Jr., B. (ed.) Advances in Cryptology – CRYPTO '97, Lecture Notes in Computer Science, vol. 1294, pp. 342–356. Springer, Berlin Heidelberg New York (August 1997)
27. Guajardo, J., Paar, C.: Itoh–Tsujii inversion in standard basis and its application in cryptography and codes. Des. Codes Cryptogr. **25**(2), 207–216 (2002)
28. Geiselmann, W., Steinwandt, R.: A redundant representation of $GF(q^n)$ for designing arithmetic circuits. IEEE Trans. Comput. **52**(7), 848–853 (2003)
29. Gutub, A.A., Tenca, A.F., Koc, C.K.: Scalable VLSI architecture for GF(p) Montgomery modular inverse computation. In: Naccache, D. (ed.) IEEE Computer Society Annual Symposium on VLSI, pp. 53–58. IEEE Computer Society Press, Los Alamitos, California (2002)
30. Guajardo Merchan, J.: Arithmetic architectures for finite fields $GF(p^m)$ with cryptographic applications. PhD thesis, Ruhr-Universität Bochum, Germany (Available at http://www.crypto.rub.de/theses.html) (July 2004)
31. Guajardo, J., Wollinger, T., Paar, C.: Area efficient $GF(p)$ architectures for $GF(p^m)$ multipliers. In: Proceedings of the 45th IEEE International Midwest Symposium on Circuits and Systems – MWSCAS 2002, Tulsa, Oklahoma, August 2002
32. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Comput. Inf. **78**, 171–177 (1988)
33. Jullien, G.A.: Residue number scaling and other operations using ROM arrays. IEEE Trans. Comput. **C-27**, 325–337 (1978)
34. Kaliski, B.S.: The montgomery inverse and its applications. IEEE Trans. Comput. **44**(8), 1064–1065 (1995)
35. Koç, Ç.K., Hung, C.Y.: Bit-level systolic arrays for modular multiplication. J. VLSI Signal Process. **3**(3), 215–223 (1991)
36. Knuth, D.E.: The Art of Computer Programming, Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading, Massachusetts (November 1971)(2nd printing)
37. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms, vol. 2, 2nd edn. Addison-Wesley, Massachussetts, USA (1973)
38. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Sov. Phys. Dokl. **7**, 595–596 (1963) (English translation)
39. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987)
40. Koblitz, N.: Hyperelliptic cryptosystems. J. Cryptology **1**(3), 129–150 (1989)
41. Koblitz, N.: An elliptic curve implementation of the finite field digital signature algorithm. In: Krawczyk, H. (ed.) Advances in Cryptology – CRYPTO 98. LNCS, vol. 1462, pp. 327–337. Springer, Berlin Heidelberg New York (1998)
42. Koren, I.: Computer Arithmetic Architectures. Prentice-Hall, New Jersey (1993)
43. Lidl, R., Niederreiter, H.: Finite fields. In: Encyclopedia of Mathematics and its Applications, vol. 20, 2nd edn. Cambridge University Press, Great Britain (1997)
44. Loidreau, P.: On the factorization of trinomials over F_3. Rapport de recherche no. 3918, INRIA (April 2000)
45. Lenstra, A., Verheul, E.: The XTR public-key cryptosystem. In: Bellare, M. (ed.) Advances in Cryptology – CRYPTO 2000. LNCS, vol. 1423, pp. 1–19. Springer, Berlin Heidelberg New York (2000)
46. Mihăilescu, P.: Optimal Galois Field Bases which are not Normal. Recent Results Session — FSE '97 (1997)
47. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) Advances in cryptology – CRYPTO '85. Lecture Notes in Computer Science, vol. 218, pp. 417–426. Springer, Berlin Heidelberg New York (August 1986)
48. Morii, M., Kasahara, M., Whiting, D.L.: Efficient bit-serial multiplication and discrete-time Wiener–Hoph equation over finite fields. IEEE Trans. Inform. Theory, **IT-35**, 1177–1184 (1989)
49. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
50. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
51. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. The CRC Press Series on Discrete Mathematics and Its Applications. CRC, Florida, USA (1997)

52. National Institute for Standards and Technology: *FIPS 186-2: Digital Signature Standard (DSS)186-2*. Gaithersburg, Maryland, USA (Available for download at http://csrc.nist.gov/encryption) (February 2000)
53. Norris, M.J., Simmons, G.J.: Algorithms for high-speed modular arithmetic. Congressus Numeratium **31**, 153–163 (1981)
54. Oo, J.Y., Kim, Y.-G., Park, D.-Y., Kim, H.-S.: Efficient multiplier architecture using optimized irreducible polynomial over $GF((3^n)^3)$. In: Proceedings of the IEEE Region 10 Conference – TENCON 99. Multimedia Technology for Asia-Pacific Information Infrastructure, vol. 1, pp. 383–386, Cheju, Korea (1999)
55. Parhami, B.: Computer Arithemtic – Algorithms and Hardware Designs. Oxford University Press, New York, USA (1999)
56. Parker, M.G., Benaissa, M.: $GF(p^m)$ multiplication using polynomial residue number systems. IEEE Trans. Circuits Syst., 2 Analog Digit. Signal Process. **42**(11), 718–721 (1995)
57. Paliouras, V., Karagianni, K., Stouraitis, T.: A low-complexity combinatorial RNS multiplier. IEEE Trans. Circuits Systems I Fund., 2 Analog Digit. Signal Process. **48**(7), 675–683 (2001)
58. Smith, P., Skinner, C.: A public-key cryptosystem and a digital signature system based on the lucas function analogue to discrete logarithms. In: Pieprzyk, J., Safavi-Naini, R. (eds.) Advances in Cryptology – ASIACRYPT'94. LNCS, vol. 917, pp. 357–364. Springer, Berlin Heidelberg New York(1995)
59. Page, D., Smart, N.P.: Hardware implementation of finite fields of characteristic three. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) Workshop on Cryptographic Hardware and Embedded Systems – CHES 2002. LNCS, vol. 2523, pp. 529–539. Springer, Berlin Heidelberg New York (2002)
60. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
61. Radhakrishnan, D., Yuan, Y.: Novel approaches to the design of VLSI RNS multipliers. IEEE Trans. Circuits Syst., 2 Analog Digit. Signal Process. **39**(1), 52–57 (1992)
62. Schneier, B.: Crypto-Gram newsletter. (available at http://www.schneier.com/crypto-gram-9805.html) May 15, 1998
63. Sloan, K.R.: Comments on a computer algorithm for calculating the product $A \cdot B$ modulo M. IEEE Trans. Comput. **C-34**(3), 290–292 (1985)
64. Smart, N.: Elliptic curve cryptosystems over small fields of odd characteristic. J. Cryptology. **12**(2), 141–151 (1999)
65. Song, L., Parhi, K.K.: Low energy digit-serial/parallel finite field multipliers. J. VLSI Signal Process. **19**(2), 149–166 (1998)
66. Soudris, D.J., Paliouras, V., Stouraitis, T., Goutis, C.E.: A VLSI design methodology for RNS full adder-based inner product architectures. IEEE Trans. Circuits Syst., 2 Analog Digit. Signal Process. **44**(4), 315–318 (1997)
67. Szabó, N., Tanaka, R.: Residue Arithmetic and its Applications to Computer Technology. McGraw-Hill, New York (1967)
68. Skavantzos, A., Taylor, F.J.: On the polynomial residue number system. IEEE Trans. Signal Process. **39**, 376–382 (1991)
69. Takagi, N.: A VLSI algorithm for modular division based on the binary GCD algorithm. In: IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, vol. E81-A, pp. 724–728 (1998)
70. Tenca, A.F., Koç, Ç.K.: A scalable architecture for montgomery multiplication. In: Koç, Ç.K., Paar, C. (eds.) Workshop on Cryptographic Hardware and Embedded Systems – CHES'99. LNCS, vol. 1717 pp. 94–108. Springer, Berlin Heidelberg New York 12–13 August 1999
71. Tawalbeh, L.A., Tenca, A.F., Park, S., Koc, C.K.: A dual-field modular division algorithm and architecture for application specific hardware. In: Thirty-Eighth Asilomar Conference on Signals, Systems, and Computers, vol. 1, pp. 483–487. Pacific Grove, California (2004)
72. von zur Gathen, J.: Irreducible trinomials over finite fields. In: Mourrain, B. (ed.) Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation – ISSAC2001, pp. 332–336. ACM, New York (2001)
73. von zur Gathen, J., Nöcker, M.: Exponentiation in finite fields: theory and practice. In: Mora, T., Mattson, H. (eds.) Applied Algebra, Agebraic Algorithms and Error Correcting Codes – AAECC-12. LNCS, vol. 1255, pp. 88–113. Springer, Berlin Heidelberg New York (2000)
74. Walter, C.D.: Logarithmic speed modular multiplication. Electron. Lett. **30**(17), 1397–1398 (1994)

75. Wang, M., Blake, I.F.: Bit serial multiplication in finite fields. SIAM J. Discrete Math. **3**(1), 140–148 (1990)
76. Wu, H., Hasan, M.A., Blake, I.F.: Low complexity parallel multiplier in F_{q^n} over F_q. IEEE Trans. Circuits Systems 1, Fund. Theory Appl. **49**(7), 1009–1013 (2002)
77. Xilinx, Inc.: The Programmable Logic Data Book (2000)
78. Zierler, N., Brillhart, J.: On primitive trinomials $(\mathrm{mod}\,2)$. Inf. Control **13**, 541–554 (1968)
79. Zierler, N., Brillhart, J.: On primitive trinomials $(\mathrm{mod}\,2)$, II. Inf. Control **14**, 566–569 (1969)
80. Zierler, N.: On $x^n + x + 1$ over $GF(2)$. Inf. Control **16**, 67–69 (1970)