Provably Sublinear Point Multiplication on Koblitz Curves and its Hardware Implementation

V.S. Dimitrov[1⋆], K.U. Järvinen[2], M.J. Jacobson, Jr.[3], W.F. Chan[3], and Z. Huang[1⋆⋆]

[1] Department of Electrical and Computer Engineering, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4, (dimitrov,huangzh)@atips.ca

[2] Signal Processing Laboratory, Helsinki University of Technology, Otakaari 5A, 02150, Espoo, Finland, kimmo.jarvinen@tkk.fi

[3] Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4, (chanwf,jacobs)@cpsc.ucalgary.ca

Abstract. We describe algorithms for point multiplication on Koblitz curves using multiple-base expansions of the form $k = \sum \pm\tau^a(\tau-1)^b$ and $k = \sum \pm\tau^a(\tau-1)^b(\tau^2-\tau-1)^c$. We prove that the number of terms in the second type is sublinear in the bit length of k, which leads to the first provably sublinear point multiplication algorithm on Koblitz curves. For the first type, we conjecture that the number of terms is sublinear and provide numerical evidence demonstrating that the number of terms is significantly less than that of τ-adic non-adjacent form expansions. We present details of an innovative FPGA implementation of our algorithm and performance data demonstrating the efficiency of our method.

1 Introduction

In 1985, Koblitz [2] and Miller [3] independently proposed the use of the additive finite abelian group of points on elliptic curves defined over a finite field for cryptographic applications. The Koblitz curves [4], or anomalous binary curves, are

$$E_a : y^2 + xy = x^3 + ax^2 + 1 \tag{1}$$

defined over $\mathbb{F}_2$, where $a \in \{0, 1\}$. The number of points on these curves when considered over $\mathbb{F}_{2^m}$ can be computed rapidly using a simple recurrence relation, and there are many prime values of m for which the number of points is twice a prime (when $a = 1$) or four times a prime (when $a = 0$). Five Koblitz curves are recommended for cryptographic use by NIST [5].

The main advantage of Koblitz curves is that the Frobenius automorphism of $\mathbb{F}_2$ acts on points via $\tau(x, y) = (x^2, y^2)$ and is essentially free to compute.

⋆ Chan, Dimitrov and Jacobson are supported in part by NSERC of Canada.

⋆⋆ This paper is extended version of our article [1] that will appear in CHES 2006.

Because τ satisfies $(\tau^2 + 2)P = \mu\tau(P)$ for all points $P \in E_a(\mathbb{F}_{2^m})$ where $\mu = (-1)^{1-a}$, we can consider τ as a complex number satisfying $x^2 - \mu x + 2 = 0$, i.e., $\tau = (\mu + \sqrt{-7})/2$. Thus, computing kP, where $k \in \mathbb{Z}$ and $P \in E_a(\mathbb{F}_{2^m})$, can be done using a representation of k involving powers of τ instead of the usual binary representation using powers of 2, yielding a point multiplication algorithm similar to the binary "double-and-add" method in which the point doublings are replaced by applications of the Frobenius [4, 6]. Solinas [6] shows how the non-adjacent form (NAF) and window-NAF methods mentioned earlier can be extended to τ-adic expansions. The resulting point multiplication algorithms require on average $(\log_2 k)/3$ point additions or $(\log_2 k)/(w+1)$ point additions using width-w window methods requiring precomputations based on P. A recent result of Avanzi et. al. [7] reduces this to $(\log_2 k)/4$ at the cost of one additional point halving, but the practicality of this method has not yet been demonstrated.

Recently, double-base integer representations have been used to devise efficient point multiplication algorithms [8–10]. For example, it can be shown that the number of terms of the form $\pm 2^a 3^b$ required to represent k is bounded by $O(\log k / \log\log k)$. These representations can be computed efficiently and the resulting point multiplication algorithms are the only known methods for which the number of required point additions is sublinear in $\log k$.

In this paper, we extend the double-base idea to τ-adic expansions for point multiplication on Koblitz curves by representing k as a sum of terms $\pm\tau^a(\tau-1)^b$. Our algorithm requires no precomputations based on the point P, no point doublings, and fewer point additions than τ-adic NAF (τ-NAF) for the five recommended Koblitz curves from [5]. Our algorithm for computing the double-base representation of k is very efficient; it requires only the unsigned τ-adic expansion of k plus a few table-lookups. A precomputed table of optimal representations for a small number of τ-adic integers is required, but these are independent of the multiplier k and the base point P. We have developed a novel FPGA implementation of both the conversion and point multiplication algorithms that demonstrates the efficiency of our method.

We conjecture that the average density of our representations is sublinear in $\log k$, and provide extensive numerical evidence showing that the density is lower than that of τ-NAF expansions. Although we do not have a proof that the number of point additions required by our algorithm is sublinear, we provide a proof that sublinearity is obtained using similar expressions involving three bases of the form $\pm\tau^a(\tau-1)^b(\tau^2-\tau-1)^c$. This work represents the first rigorously-proven sublinear point multiplication algorithm using complex bases.

Avanzi and Sica [11] have independently investigated applying the double-base idea to Koblitz curves using bases τ and 3. It is not clear how their algorithm performs in practice. They include a proof that the density of their multiplier representations is sublinear under the unproven but reasonable assumption that the irrationality measure of $\log_2 3$ and $\arg(\tau)/\pi$ is 2, but his proof has been shown to have a gap [12].

The remainder of the paper is organized as follows. In Sec. 2 we present our provably sublinear point multiplication algorithm. We present a similar algorithm using only two complex bases in Sec. 3. Although we cannot prove sublinearity for this algorithm, we conjecture that the density of the representations is in fact sublinear, and provide numerical evidence in Subsection 3.2 indicating that the density of our representations is lower than that of τ-NAF representations. A description of our FPGA implementation and numerical data demonstrating its efficiency are presented in Sec. 4. Finally, we conclude with an outlook on possible directions for further research.

2 Multi-dimensional Frobenius Expansions

We start with the following two definitions:

Definition 1. *A complex number, ξ of the form $e + f\tau$, e, f-integers is called a Kleinian integer [13].*

Definition 2. *A Kleinian integer ω of the form $\omega = \pm\tau^x(\tau-1)^y$, $x, y \geq 0$ is called a $\{\tau, \tau-1\}$-Kleinian integer.*

Definition 3. *A Kleinian integer ω of the form $\omega = \pm\tau^x(\tau-1)^y(\tau^2-\tau-1)^z$, $x, y, z \geq 0$ is called a $\{\tau, \tau-1, \tau^2-\tau-1\}$-Kleinian integer.*

The main idea of the new point multiplication algorithm over Koblitz curves is to extend the existing and widely-used τ-NAF expansion of the scalar to a new form which will speed up the computations. The improvements obtained in the paper are based on the following representation, which we will call two-dimensional or three-dimensional Frobenius expansions (or $\{\tau, \tau-1\}$-expansion and $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansion, for short):

$$k = \sum_{i=1}^{d} s_i \tau^{a_i} (\tau-1)^{b_i}, \quad s_i = \pm 1, \quad a_i, b_i \in \mathbb{Z}_{\geq 0}, \tag{2}$$

$$k = \sum_{i=1}^{d} s_i \tau^{a_i} (\tau-1)^{b_i} (\tau^2-\tau-1)^{c_i}, \quad s_i = \pm 1, \quad a_i, b_i, c_i \in \mathbb{Z}_{\geq 0} \ . \tag{3}$$

Such representations are clearly highly redundant. If we rearrange the summands in the above formula, then, using two bases, we can represent the scalar k as

$$k = \sum_{l=1}^{\max(b_i)} (\tau-1)^l \left(\sum_{i=1}^{\max(a_{i,l})} s_{i,l} \tau^{a_{i,l}} \right) \tag{4}$$

where $\max(a_{i,l})$ is the maximal power of τ that is multiplied by $(\tau-1)^l$ in (2). Using three bases, we can represent k as

$$k = \sum_{l_2}^{\max c_i} (\tau^2-\tau-1)^l_2 \sum_{l_1=1}^{\max(b_i)} (\tau-1)^{l_1} \left(\sum_{i=1}^{\max(a_{i,l_1,l_2})} s_{i,l_1,l_2} \tau^{a_{i,l_1,l_2}} \right) \tag{5}$$

where $\max(a_{i,l_1,l_2})$ is the maximal power of τ that is multiplied by $(\tau-1)^{l_1}(\tau^2-\tau-1)^{l_2}$ in (3).

Alg. 1 computes kP given a $\{\tau, \tau-1\}$-expansion of k. The corresponding algorithm for $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansions will be described later, along with a proof that the number of point additions is sublinear in $\log k$. Essentially, kP

Algorithm 1 Point multiplication using $\{\tau, \tau-1\}$-expansions.

INPUT: k, P
OUTPUT: $Q = kP$
  $P_0 \leftarrow P$
  $Q \leftarrow \mathcal{O}$
  for $i = 0$ to j **do**
    $S \leftarrow r_i(k)P_i$ {One dimensional τ-NAF corresponding to $(\tau-1)^l$ in (4)}
    $P_{i+1} \leftarrow \tau P_i - P_i$
    $Q \leftarrow Q + S$

is computed via a succession of one-dimensional τ-adic expansions.

The representation of k given in (4) is the cornerstone of our algorithm, so some comments on it are in order.

1. The multiplications by $\tau-1$ cost one Frobenius mapping (free in our computational model) and one point subtraction. The multiplications by $\tau^2-\tau-1$ cost two Frobenius mappings and two point subtractions. Therefore, the total number of point additions/subtractions, $AS(k)$, is given by
$$AS(k) = \max(b_i) + d - 1$$
in the case of $\{\tau, \tau-1\}$-expansions and
$$AS(k) = d + \max(b_i)\max(c_i) - 1$$
in the case of $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansions. The smallest possible value of $\max(b_i)$ and $\max(c_i)$, 0, corresponds to the classical (one-dimensional) τ-NAF expansion, for which it is known that the expected number of point additions/subtractions is $(\log_2 k)/3$. It is clear that by allowing larger values for $\max(b_i)$ and $\max(c_i)$ one would decrease the corresponding number of summands, d. Therefore, it is vital to find out the optimal values for $\max(b_i)$ as a function of the size of the scalar.

2. Finding an algorithm that can return a fairly short decomposition of k as the sum of $\{\tau, \tau-1\}$-Kleinian integers is absolutely essential. The most straightforward idea seems to be the greedy algorithm described in Alg. 2. A greedy algorithm for computing $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansions is an easy generalization of this algorithm.

The complexity of the greedy algorithm depends crucially on the time spent to find the closest $\{\tau, \tau-1\}$-Kleinian integer to the current Kleinian integer.

Algorithm 2 Greedy algorithm for computing $\{\tau, \tau-1\}$-expansions.

INPUT: A Kleinian integer $\xi = e + f\tau$
OUTPUT: $\{\omega_1, \ldots, \omega_d\}$, a $\{\tau, \tau-1\}$-expansion of ξ

```
i ← 1
while ξ ≠ 0 do
    Find ω_i = ±τ^{a_i}(τ − 1)^{b_i}, a_i, b_i ≥ 0, the closest {τ, τ − 1}-Kleinian integer to ξ.
    ξ ← ξ − ω_i
    i ← i + 1
```

Unfortunately we were not able to find a significantly more efficient method to do this than precomputing all Kleinian integers $\pm\tau^x(\tau-1)^y$ for x, y less than certain bounds and finding the closest one using exhaustive search. In the next subsection, we present an efficient algorithm for computing $\{\tau, \tau-1\}$-expansions with slightly more weight than those produced by the greedy algorithm and an algorithm for computing $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansions with weight provably sublinear in $\log k$.

2.1 Comparison to Double-base Number Systems

The similarities between (2) and the double-base number system (DBNS), in which one represents integers as the sum or difference of numbers of the form $2^a 3^b$, a, b non-negative integers (called $\{2, 3\}$-integers), are apparent. In the case of DBNS, one can prove the following result:

Theorem 1. *Every positive integer, n, can be written as the sum of at most $O(\log n/\log\log n)$ $\{2, 3\}$-integers and (one) such representation can be found by using the greedy algorithm.*

The key point in proving this theorem is the following result of Tijdeman [14].

Theorem 2. *Let x and y be two $\{2, 3\}$-integers, $x > y$. Then there exist effectively computable constants, c_1 and c_2, such that*

$$\frac{x}{(\log x)^{c_1}} < x - y < \frac{x}{(\log x)^{c_2}} \ .$$

The proof of Theorem 1 uses only the first inequality.

Theorem 2 provides a very accurate description of the difference between two consecutive $\{2, 3\}$-integers. More to the point, it can be generalized easily to any set of $\{p_1, p_2, \cdots, p_s\}$-integers if p_s is fixed. The proof depends on the main result of [15] from the theory of linear form in logarithms.

Theorem 3. *Let $a_1, a_2, \cdots, a_k$ be nonzero algebraic integers and $b_1, b_2, \cdots, b_k$ rational integers. Assume $a_1^{b_1} a_2^{b_2} \cdots a_k^{b_k} \neq 1$ and $B = \max(b_1, b_2, \cdots, b_k)$. Then the following inequality holds:*

$$\left| a_1^{b_1} a_2^{b_2} \cdots a_k^{b_k} - 1 \right| \geq \exp(-C(k) \log a_1 \log a_2 \cdots \log a_k)$$

where $C(k) = \exp(4k + 10k^{3k+5})$.

The constant $C(k)$ is huge, even in the case of linear forms in two logarithms, approximately $\exp(6 \cdot 10^9)$, By using some results aimed specifically at the case of two logarithms [16], one can reduce $C(k)$ to $\exp(10^7)$, but this is still enormous. However, practical simulations suggest that this constant is likely to be much smaller, perhaps less than 100.

There are two very essential points that are often overlooked in the formulations of the above theorems [17]:

1. the estimates are correct if the algebraic numbers used are real,
2. if the algebraic numbers are complex, then the estimates provided remain unchanged if one of them, say a_1, has an absolute value strictly greater than absolute values of the other algebraic numbers.

The latter point is what prevents us from applying Tijdeman's Theorem 2 to the case of $a_1 = \tau, a_2 = \tau - 1$. Thus, we are not in position to trivially extend the proof of Theorem 1 to the case of $\{\tau, \tau - 1\}$-expansions of Kleinian integers. Nevertheless, extensive numerical simulations (by using several attempted optimizations of Alg. 2) has led us to the following conjecture:

Conjecture 1. Every Kleinian integer, $\xi = a + b\tau$, can be represented as the sum of at most $O(\log N(\xi)/\log\log N(\xi))$ $\{\tau, \tau - 1\}$-Kleinian integers, where $N(\xi)$ is the norm of ξ.

A very recent paper by Avanzi and Sica [11] contains a proof that Conjecture 1 is true if one uses $\{\tau, 3\}$-Kleinian integers under the unproven but reasonable assumption that the irrationality measure of $\log_2 3$ and $\arg(\tau)/\pi$ is 2. Unfortunately, the proof, even with the assumption on irrationality measures, has a gap [12]. The use of two complex bases, used in this paper, increases the theoretical difficulties in proving the conjecture, but provides much more practical algorithms.

However, in the case of *three* bases we can prove without any assumptions the following:

Theorem 4. *Every Kleinian integer $\zeta = a + b\tau$ can be represented as the sum of at most $O(\log N(\zeta)/(\log\log N(\zeta)))$ $\{\tau, \tau - 1, \tau^2 - \tau - 1\}$-Kleinian integers, such that the largest power of both $\tau - 1$ and $\tau^2 - \tau - 1$ is $O(\log^\alpha N(\zeta))$ for any real constant α where $0 < \alpha < 1/2$.*

Proof. We assume that $b = 0$; otherwise, one applies the same proof for the real and imaginary part of ζ, which leads to doubling the implicit constant hidden in the big-O notation.

Let α be a real constant where $0 < \alpha < 1/2$. We determine the τ-adic representation of a, the real part of ζ, using digits 0 and 1. The length of this expansion is $O(\log N(\zeta))$. We break this representation into $\lceil \log^{1-\alpha} N(\zeta) \rceil$ blocks, where each block contains $O(\log^\alpha N(\zeta))$ digits. Each of these blocks corresponds to a Kleinian integer $c_i + d_i\tau$, $i = 0, 1, \ldots, \lceil \log^{1-\alpha} N(\zeta) \rceil$, where the size of both c_i and d_i is $O(\log^\alpha N(\zeta))$. Now, we represent each integer c_i and d_i in double-base representation using bases 2 and 3. According to Theorem 1, these numbers will require at most

$$O\left(\log^\alpha N(\zeta)/(\log\log^\alpha N(\zeta))\right) = O\left(\log^\alpha N(\zeta)/(\log\log N(\zeta))\right)$$

summands of the form $2^x 3^y$ where $x, y \geq 0$ and $x, y \in O(\log^\alpha N(\zeta))$. Using the fact that $2 = \tau(1-\tau)$ and $3 = 1 - \tau - \tau^2$, we substitute the 2's and 3's in the 2, 3-expansions of c_i and d_i to obtain $\{\tau, \tau-1, \tau^2-\tau-1\}$-Kleinian integer expansions of each $c_i + d_i\tau$, $i = 0, 1, \ldots, \lceil \log^{1-\alpha} N(\zeta) \rceil$. To obtain the expansion of $\zeta = a + b\tau$, we multiply each term of the form $\pm\tau^x(\tau-1)^y(\tau^2-\tau-1)^z$ by τ^i where i is the index of the corresponding block. Note that $x, y, z \in O(\log^\alpha N(\zeta))$. Since the number of blocks is $\lceil \log^{1-\alpha} N(\zeta) \rceil$ and each block requires $O(\log^\alpha N(\zeta)/(\log\log N(\zeta))$ $\{\tau, \tau-1, \tau^2-\tau-1\}$-Kleinian integers, we conclude that the overall number of Kleinian integers used to represent ζ is

$$O\left(\frac{\log^\alpha N(\zeta)}{\log\log N(\zeta)} \log^{1-\alpha} N(\zeta)\right) = O\left(\frac{\log N(\zeta)}{\log\log N(\zeta)}\right) .$$

The exponents of $\tau - 1$ and $\tau^2 - \tau - 1$ are bounded by $O(\log^\alpha N(\zeta))$. □

Theorem 4 is in fact constructive and leads to the following sublinear point multiplication algorithm (Algorithm 3).

Algorithm 3 Point multiplication algorithm using $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansions.

INPUT: An Kleinian integer ζ, a point P on a Koblitz curve, a real constant α with $0 < \alpha < 1/2$
OUTPUT: $Q = \zeta P$
Compute in succession for $i = 0, 1, \ldots, \lceil \log^\alpha N(\zeta) \rceil$ the points $P_i^{(1)} = (\tau-1)P_{i-1}^{(1)}$ and $P_i^{(2)} = (\tau^2-\tau-1)P_{i-1}^{(2)}$ where $P_0^{(1)} = P_0^{(2)} = P$.
Compute the points $Q_{i_1,i_2} = P_{i_1}^{(1)} + P_{i_2}^{(2)}$ for $i_i, i_2 = 0, 1, \ldots, \lceil \log^\alpha N(\zeta) \rceil$.
Compute a $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansion of the form (5) using Theorem 4.
Apply in succession τ-NAF based point multiplications based on (5) to compute Q.

The analysis of Alg. 3 is simple. Step 1 requires $O(\log^\alpha N(\zeta))$ point additions and Step 2 requires $O(\log^{2\alpha} N(\zeta))$ point additions. Because $\alpha < 1/2$, the total number of point additions for Steps 1 and 2 is $o(\log N(\zeta))$. According to Theorem 4, Step 3 requires $O(\log N(\zeta)/(\log\log N(\zeta)))$ point additions. The total number of point additions for Alg. 3 is therefore $O(\log N(\zeta)/(\log\log N(\zeta)))$. Thus, one can compute kP in $O(\log k/(\log\log k)$ point additions by computing $\zeta \equiv k \pmod{(\tau^m-1)/(\tau-1)}$ and applying Alg. 3 to compute ζP.

Note that the first two steps of Alg. 3 are independent of k. If a fixed base point P is to be used, the points Q_{i_1,i_2} may be precomputed.

The parameter α can be chosen in a variety of ways. The total number of point additions required by all three steps is roughly $\log^\alpha N(\zeta) + \log^{2\alpha} N(\zeta) +$

$2\log N(\zeta)/(\alpha \log\log N(\zeta))$; for $163 < N(\zeta) < 571$, taking α such that $0.365 < \alpha < 0.368$ minimizes this quantity. Smaller values of α reduce the number of points Q_{i_1,i_2} that must be precomputed and stored at the cost of increasing the number of point additions that must be performed in Step 3. On the other hand, larger values of α decrease the number of point additions in Step 3 at the cost of having to precompute and store more points.

3 A Practical Blocking Algorithm

Although, as proved in Theorem 4, using $\{\tau, \tau-1, \tau^2-\tau-1\}$-expansions does lead to a sublinear point multiplication algorithm, the resulting algorithm is likely not suitable for practical purposes. Nevertheless, assuming the truth of Conjecture 1, we can devise an efficient algorithm that computes $\{\tau, \tau-1\}$-expansions with sublinear density of Kleinian integers. This algorithm is based on the following theorem.

Theorem 5. *Assuming Conjecture 1, every Kleinian integer, $\xi = a + b\tau$, can be represented as the sum of at most $O\left(\log N(\xi)/\log\log N(\xi)\right)$ $\{\tau, \tau-1\}$-Kleinian integers such that the largest power of $\tau - 1$ is $O\left(\log N(\xi)/\log\log N(\xi)\right)$.*

Proof. We shall assume that $b = 0$; otherwise, one applies the same proof for the real and imaginary part of ξ, which leads to doubling the implicit constant hidden in the big-O notation. Next, we represent a, the real part of ξ, in base-τ expansion with digits 0 and 1. The length of this expansion is $O(\log N(\xi))$. Now we break this representation into $\log\log N(\xi)$ blocks, each of length $O\left(\log N(\xi)/\log\log N(\xi)\right)$. Every block of digits corresponds to a Kleinian integer and, if Conjecture 1 is true, then every block can be represented as the sum at most

$$\begin{aligned} & O\left(\frac{\log N(\xi)/\log\log N(\xi)}{\log\left(\log N(\xi)/\log\log N(\xi)\right)}\right) \\ = \; & O\left(\frac{\log N(\xi)}{\log^2 \log N(\xi) - \log\log N(\xi) \log\log\log N(\xi)}\right) \\ = \; & O\left(\frac{\log N(\xi)}{\log^2 \log N(\xi)}\right) \end{aligned}$$

$\{\tau, \tau-1\}$-Kleinian integers. As the number of blocks is $\log\log N(\xi)$, we conclude that this representation consists of at most $O\left(\log N(\xi)/\log\log N(\xi)\right)$ $\{\tau, \tau-1\}$-Kleinian integers. The highest power of $\tau - 1$ that can occur in such a representation is governed by the highest power that can occur in every block and it is at most $O\left(\log N(\xi)/\log\log N(\xi)\right)$. □

This theorem leads to an efficient method to compute $\{\tau, \tau-1\}$-expansions with sublinear density under Conjecture 1. The idea, described in Alg. 4, is to apply the blocking strategy described in the proof and compute optimal $\{\tau, \tau - 1\}$-expansions for each block.

There are four important points regarding the implementation Alg. 4:

Algorithm 4 Blocking algorithm for computing $\{\tau, \tau - 1\}$-expansions.

INPUT: A Kleinian integer $\xi = e + f\tau$, block size w, precomputed table of the minimal $\{\tau, \tau - 1\}$-expansion of every Kleinian integer $\sum_{i=0}^{w-1} d_i\tau^i$, $d_i \in \{0, 1\}$
OUTPUT: List L of $\{\tau, \tau - 1\}$-Kleinian integers representing $\{\tau, \tau - 1\}$-expansion of ξ

$L = \emptyset$
Compute the τ-adic expansion of $\xi = \sum_{i=0}^{l} d_i\tau^i$, $d_i \in \{0, 1\}$
for $j = 0$ to $\lceil l/w \rceil$ **do**
  {Process blocks of length w}
  Find minimal $\{\tau, \tau - 1\}$-expansion of $\sum_{i=0}^{w-1} d_{i+jw}\tau^i$ from the precomputed table
  Multiply each term of the expansion by τ^{jw} and add to L

1. All powers of τ can be reduced modulo m, as $(\tau^m)P = P$ for all $P \in E_a(\mathbb{F}_{2^m})$.
2. The bit-string $d_{w-1} \dots d_1d_0$ corresponding to any block can be used as an index into the table of minimal $\{\tau, \tau - 1\}$-expansions.
3. One can choose the size of the blocks based on available memory. The larger the block size, the lower the density of the $\{\tau, \tau - 1\}$-expansions produced.
4. If the block size is not too big, one can precompute the minimal $\{\tau, \tau - 1\}$-expansion of every Kleinian integer of the form $\sum_{i=0}^{w-1} d_i\tau^i$, $d_i \in \{0, 1\}$, thereby ensuring as low a density as possible. This precomputation can be done using exhaustive search and need only be done once per elliptic curve, as it does not depend on k nor the base point P.

3.1 Example

Consider the representation of 6465 into a $\{\tau, \tau - 1\}$-expansion by using the two different algorithm we have described. Assume that we intend to compute $(6465)P$ for some point $P \in E_1(\mathbb{F}_{2^{163}})$, so $\tau = (1 + \sqrt{-7})/2$. As in the case of computing the τ-NAF expansion, we first do a partial reduction of 6465 modulo $(\tau^{163} - 1)/(\tau - 1)$ as in [6], yielding $\xi = -104 + 50\tau$. The greedy algorithm, Alg. 2, returns

$$\xi = \tau^8(\tau - 1) + \tau^2(\tau - 1)^2 + \tau^8(\tau - 1)^6 \quad .$$

The blocking algorithm, Alg. 4, using a block size $w = 7$ returns the same representation.

Fig. 1 illustrates the idea behind the blocking algorithm. The τ-adic expansion of ξ is $\tau^2 + \tau^5 + \tau^9 + \tau^{11} + \tau^{12} + \tau^{13}$. This 14-bit expansion of ξ is broken into two 7-bit blocks. The right block corresponds to $\tau^2 + \tau^5$, and $\tau^2(\tau - 1)^2$ is its optimal $\{\tau, \tau - 1\}$-expansion. The left block corresponds to $\tau^2 + \tau^4 + \tau^5 + \tau^6$, and $\tau(\tau - 1) + \tau(\tau - 1)^6$ is its optimal expansion. Finally, multiplying the expression for the left block by τ^7 yields the $\{\tau, \tau - 1\}$-expansion of ξ.

The usefulness of this idea is perhaps clearly visible if one uses a geometric representation of the $\{\tau . \tau - 1\}$-expansion. Table 1 shows the two-dimensional representation of the $\{\tau, \tau - 1\}$-expansion computed above. The expansion is very sparse — of the 63 possible terms that could occur, assuming τ^8 is the

τ-adic expansion:	1 1 1 0 1 0 0 0 1 0 0 1 0 0	
	$\Downarrow$	$\Downarrow$
Segment size = 7 :	1 1 1 0 1 0 0	0 1 0 0 1 0 0
	$\Downarrow$	$\Downarrow$
Table look-up:	$\tau(\tau-1)+\tau(\tau-1)^6$	$\tau^2(\tau-1)^2$
	$\Downarrow$	$\Downarrow$
$\{\tau,\tau-1\}$-expansion:	$\tau^7[\tau(\tau-1)+\tau(\tau-1)^6]+\tau^2(\tau-1)^2$	

Fig. 1. Expansion of $-104+50\tau$ using Alg. 4.

Table 1. The 2-dimensional pictorial representation of the $\{\tau,\tau-1\}$-expansion of 6465 obtained by both Alg. 2 and Alg. 4.

	τ^0	τ^1	τ^2	τ^3	τ^4	τ^5	τ^6	τ^7	τ^8
$(\tau-1)^0$									
$(\tau-1)^1$									+1
$(\tau-1)^2$			+1						
$(\tau-1)^3$									
$(\tau-1)^4$									
$(\tau-1)^5$									
$(\tau-1)^6$									+1

maximum power of τ and $(\tau-1)^6$ is the maximum power of $\tau-1$, only three actually occur in the expansion. Furthermore, note that when computing kP using this representation, each row in the table represents a one-dimensional τ-adic expansion, and that these are very sparse.

3.2 Numerical Evidence

In this section, we present results from software implementations of Alg. 2 and Alg. 4, the greedy and blocking-based $\{\tau,\tau-1\}$-expansion algorithms. The objective is to compare the density of the $\{\tau,\tau-1\}$-expansions computed by our algorithms with τ-NAF expansions. Our algorithms and the algorithm for computing the τ-NAF [6] of k were implemented in C, using the GMP library for multi-precision integer arithmetic. Tests were run on an Intel Xeon 2.8 GHz CPU running Linux.

In Table 2 we present the average time required to compute the $\{\tau,\tau-1\}$-expansion of k compared with the time to compute the τ-NAF of k for the sizes of k corresponding to the five Koblitz curves recommended by NIST [5]. We used the blocking algorithm Alg. 4 with $w=10$ and $\max(b_i)=6$ to compute the $(\tau,\tau-1)$-expansions, and 500000 random values of k for each size. Our implementation of Alg. 4 is faster than that for computing the τ-NAF, by a factor of up to 20%.

Theorem 5 states that our conversion algorithm outputs expansions of k with sublinear density even if the maximum power of $\tau-1$ is bounded by some constant $\max(b_i)$ as long as any sublinear expansion exists. For practical purposes,

Table 2. Average time (in μ-seconds) to compute representations of k.

$\log_2 k$	τ-NAF	$\{\tau, \tau-1\}$
163	60.0	51.9
233	83.5	66.2
283	101.2	82.2
409	150.6	121.0
571	218.4	176.8

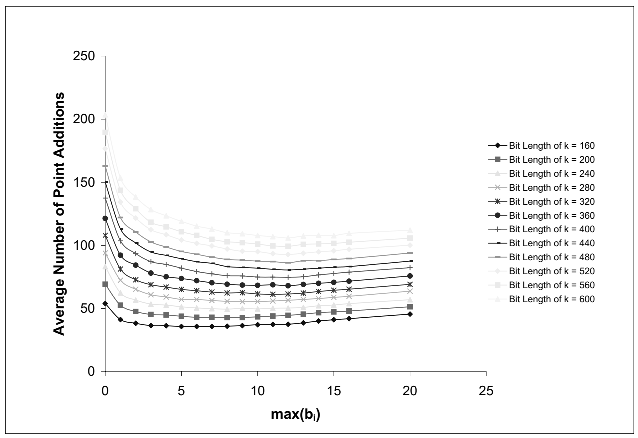

Fig. 2. Average number of point additions to compute kP as a function of $\max(b_i)$.

we need to know what value of $\max(b_i)$ gives us minimal weight expansions on average.

In Fig. 2 we plot the average number of point additions required to compute kP using a $\{\tau, \tau-1\}$-expansion of k, i.e., the number of non-zero terms in the expansion plus $\max(b_i) - 1$, as a function of $\max(b_i)$. These average values were computed using 1000 random values of k of each size. This graph illustrates that each size of k has an optimal value of $\max(b_i)$ ranging from 4 to 12. The precise value of $\max(b_i)$ to use corresponds to minimizing the overall time required to compute kP given a $\{\tau, \tau-1\}$-expansion. As shown in Sec. 4.4, $\max(b_i) = 3$ turns out to be optimal for our FPGA implementation of point multiplication on $E_1(\mathbb{F}_{2^{163}})$.

In Table 3 we list the average number of point additions required to compute kP on the five NIST-recommended Koblitz curves [5] when using τ-NAF, our greedy $\{\tau, \tau-1\}$-expansion algorithm (Alg. 2), and our blocking-based Alg. 4 using block lengths of $w = 5, 10, 16$ and $\max(b_i) = 6$. In all cases the data are taken as the average over 500000 random values of k. Our algorithm requires significantly fewer point additions than τ-NAF in all cases.

Table 3. The average number of point additions required to compute kP for the five Koblitz curves in [5].

		Alg. 2	Alg. 4 (blocking)		
$\log_2 k$	τ-NAF	(greedy)	$w = 5$	$w = 10$	$w = 16$
163	54.25	36.37	47.86	40.00	37.22
233	77.59	49.31	66.23	54.96	50.76
283	94.25	58.64	79.37	65.66	60.49
409	137.12	81.84	113.64	93.63	85.68
571	190.25	111.90	154.98	127.21	117.04

4 FPGA Implementation

An FPGA implementation was designed in order to investigate the performance of the new algorithm in practice. The design implements kP on the NIST curve K-163 defined by (1), where $a = 1$, over $\mathbb{F}_{2^{163}}$ [5].

As the number of zero coefficients in a $\{\tau, \tau - 1\}$-expansion is large, a normal basis $\mathbb{F}_{2^m}$ was selected. In a normal basis, an element $A \in \mathbb{F}_{2^m}$ is represented as $A = \sum_{i=0}^{m-1} a_i \alpha^{2^i}$ where $a_i \in \{0, 1\}$ and $\alpha^{2^i} \neq \alpha^{2^j}$ for all $i \neq j$ and $\alpha^{2^m} = \alpha$. Thus, it is obvious that the squaring operation A^2 is a cyclic right shift of the bit vector $(a_0, a_1, \ldots, a_{m-1})$ which is fast if implemented in hardware.

Affine coordinates, $\mathcal{A}$, and López-Dahab coordinates, $\mathcal{LD}$ [18], are used for representing points on $E_a(\mathbb{F}_{2^m})$. In $\mathcal{A}$, a point P is represented with two coordinates as $P = (x, y)$, and, in $\mathcal{LD}$, with three coordinates as $P = (X, Y, Z)$. The $\mathcal{LD}$ triple represents the point $(X/Z, Y/Z^2)$ in $\mathcal{A}$ [18]. Three elements $x, y, \bar{y} = x + y \in \mathbb{F}_{2^m}$ are required to represent P and $-P$ in $\mathcal{A}$. A point addition in $\mathcal{A}$ is performed as presented, e.g., in [19], and its cost is $\mathsf{I} + 2\mathsf{M} + \mathsf{S} + 8\mathsf{A}$ where I, M, S, and A denote inversion, multiplication, squaring, and addition in $\mathbb{F}_{2^m}$, respectively. A point addition in $\mathcal{LD}$ is performed as presented in [20], and it requires $13\mathsf{M} + 4\mathsf{S} + 9\mathsf{A}$. A point addition $Q + P$, where Q is in $\mathcal{LD}$ and P in $\mathcal{A}$, is called the mixed coordinate point addition, and it requires only $9\mathsf{M}+5\mathsf{S}+9\mathsf{A}$ [21]. If the curve is fixed and both P and $-P$ are available, one multiplication and one addition can be omitted resulting $8\mathsf{M} + 5\mathsf{S} + 8\mathsf{A}$. The $\mathcal{A} \mapsto \mathcal{LD}$ mapping does not require any operations in $\mathbb{F}_{2^m}$ while $\mathcal{LD} \mapsto \mathcal{A}$ requires $\mathsf{I} + 2\mathsf{M} + \mathsf{S}$. The cost of a Frobenius mapping is $3\mathsf{S}$ in $\mathcal{LD}$ and $2\mathsf{S}$ in $\mathcal{A}$. An inversion in $\mathbb{F}_{2^m}$ is computed as suggested by Itoh and Tsujii in [22]. The Itoh-Tsujii inversion requires $m - 1$ squarings and $\lfloor \log_2(m - 1) \rfloor + H_w(m - 1) - 1$ multiplications, where $H_w(m - 1)$ is the Hamming weight of $m - 1$ [22]. Hence, $\mathsf{I} = 9\mathsf{M} + 162\mathsf{S}$ if $m = 163$.

Different coordinates are used in Alg. 1 as follows: the point addition in $\mathcal{A}$ is used in computing P_i so that the point addition in mixed coordinates can be used in $S \leftarrow S \pm P_i$ computations. Because the results of row multiplications are in $\mathcal{LD}$, the point addition in $\mathcal{LD}$ must be used for $Q \leftarrow Q + S$ computations.

The implementation was designed especially for Xilinx Virtex-II family FPGAs. The implementation includes a field arithmetic processor (FAP) for arithmetic in $\mathbb{F}_{2^m}$, control logic for controlling the FAP, and a converter for converting k to a $\{\tau, \tau - 1\}$-expansion. The FAP is considered in Sec. 4.1, the control logic in Sec. 4.2, and the conversion unit in Sec. 4.3.

4.1 Field Arithmetic Processor (FAP)

The FAP includes a multiplier, a squarer, an adder, a storage element and control logic. A storage element for m-bit elements of $\mathbb{F}_{2^m}$ is required in order to store points and temporary variables during computation of kP. As Xilinx devices offer embedded memory blocks which can be used without consuming logic resources, the storage element is implemented in BlockRAMs. One dual-port BlockRAM can be configured into a 512×36-bit mode. All m bits of an element must be accessed in parallel in the FAP architecture. Hence, $\lceil \frac{m}{36} \rceil = 5$ BlockRAMs are required. Write and read operations require both one clock cycle, i.e. $\mathsf{W} = \mathsf{R} = 1$.

The squarer is a shifter which is capable of performing operations A^{2^d}, where $A \in \mathbb{F}_{2^m}$ and $0 \leq d \leq d_{\max} = 2^6 - 1$,. Thus, A^{2^d} operations can be performed with a cost of S. Addition in $\mathbb{F}_{2^m}$ is simply a bitwise exclusive-or (xor). Both squaring and addition are performed in one clock cycle, i.e. $\mathsf{S} = \mathsf{A} = 1$.

Field multiplication is critical for the overall performance. The multiplier is a digit-serial implementation of the Massey-Omura multiplier [23]. In a bit-serial Massey-Omura multiplier, one bit of the output is calculated in one clock cycle and, hence, m cycles are required in total. One bit c_i of the result $C = A \times B$ is computed from A and B by using an F-function. The F-function is field specific, and the same F is used for all output bits c_i as follows: $c_i = F(A_{\lll i}, B_{\lll i})$, where $\lll i$ denotes cyclical left shift by i bits. [5, 23]

In a digit-serial implementation, D bits are computed in parallel. Hence, $\lceil \frac{m}{D} \rceil$ cycles are required in one multiplication. In this FAP, $D = 24$. The F-function is pipelined in order to increase clock frequency by adding one register stage. As loading the operands into the shift registers requires one clock cycle and pipelining increases latency by one clock cycle, the latency is $\mathsf{M} = \lceil \frac{163}{24} \rceil + 2 = 9$.

4.2 Control Logic

Logic controlling the FAP consists of a storage for k, a control finite state machine (FSM) and a ROM for control sequences.

The implementation handles k in a coded form. The coding is performed using $\kappa : \{s, d\}$ symbols, where $s \in \{0, \bar{0}, 1, \bar{1}\}$ and $0 \leq d \leq d_{\max}$. $\bar{0}$ is a symbol reserved for a row change not -0. Coding is started from the first non-zero signed bit of the first row and it proceeds as follows: s is the signed bit starting a symbol and d is the number of Frobenius mappings following s, i.e. the number of consecutive zeros plus one (the Frobenius map associated with the start bit of the next symbol). If the run of consecutive Frobenius maps is longer than $d_{\max}$, the run must be divided into two symbols and, for the latter one, $s = 0$. Each κ, with the exception of the row change symbol, transforms into an operation $S \leftarrow \tau^d(S + sP)$ on $E_a(\mathbb{F}_{2^m})$. Let $Z(k)$ denote the maximum number of consecutive Frobenius mappings required by k. Then, the number of κ-symbols, e, required to represent k, is given by $e \geq H_w(k) + j$, with equality if and only if $d_{\max} \geq Z(k)$.

Control FSM takes κ-symbols as input and, according to s and d of κ, it sets addresses of the control sequence ROM. The control sequences controlling the FAP consist of successive FAP instructions directly controlling the FAP. There are separate control sequences for $P_{i+1} \leftarrow \tau P_i - P_i$ computation (Frobenius map and point addition in $\mathcal{A}$), point addition and subtraction (point addition in the mixed coordinates), Frobenius map, row change (point addition in $\mathcal{LD}$), and $\mathcal{LD} \mapsto \mathcal{A}$ mapping. They are all stored in a ROM implemented in a BlockRAM.

If implemented so that, for each operation, the operands would be first read from the memory, then the operation calculated and finally the result saved to the memory, the latency of an operation would be the latency of the operation (M, S or A) plus two clock cycles ($\mathsf{R} + \mathsf{W}$). However, different operations can be performed with the same operands without reading the operands more than once, and the operands of the next operation can be read while the previous operation is performed if the result is not required in the next operation. When the control sequences were carefully hand-optimized, different operations have the following latencies: point addition in the mixed coordinates $\mathsf{L}_{\mathcal{M}} = 98$, the Frobenius map $\mathsf{L}_F = 6$, row change (point addition in $\mathcal{LD}$) $\mathsf{L}_{\mathcal{LD}} = 153$, the computation of P_i $\mathsf{L}_{P_i} = 182$, and the $\mathcal{LD} \mapsto \mathcal{A}$ mapping $\mathsf{L}_{\mathcal{LD} \mapsto \mathcal{A}} = 160$. The first point addition of each row is simply $S \leftarrow \pm P_i$ and the first row change operation is given by $Q \leftarrow S$. Both of these operations have a latency of $\mathsf{L}_C = 6$. In the beginning, an initialization including, e.g., the transferring of P into the FAP, needs to be performed. The latency of the initialization is $\mathsf{L}_I = 10$. Thus, it follows that the latency of the computation of kP becomes

$$\mathsf{L}_{kP} = (H_w(k) - (j+1))\mathsf{L}_{\mathcal{M}} + (j+2)\mathsf{L}_C + (e-j)\mathsf{L}_F + j(\mathsf{L}_{\mathcal{LD}} + \mathsf{L}_{P_i}) + \mathsf{L}_I + \mathsf{L}_{\mathcal{LD}\mapsto\mathcal{A}} \quad (6)$$

and, by assuming that $d_{\max} \geq Z(k)$, i.e. $e = H_w(k) + j$, (6) simplifies to

$$\mathsf{L}_{kP} = 104\, H_w(k) + 243\, j + 84. \quad (7)$$

4.3 Conversion Unit

The conversion unit, which converts an integer k into a $\{\tau, \tau - 1\}$-expansion, is a straightforward implementation of Alg. 4, our blocking-based method.

The main part of this unit is an ALU, which has two integer multipliers, each of which makes use of one 18-bit by 18-bit embedded multiplier to create 102-bit by 102-bit products. The ALU also includes adders, shifters and the rounding function required by the partial reduction algorithm [6]. The conversion unit uses the ALU and two intermediate registers for reducing every integer k to an equivalent $r_0 + r_1\tau$, then gets the τ-adic expansion by a shift-and-add circuit, which produces one bit per cycle, from the least significant bit to the most significant bit.

For our implementation, we selected a block size of 10, so every 10 bits of the τ-adic expansion are used as an index into a look-up table. This table has one entry for each possible index $(b_9 b_8 \ldots b_0)$, $b_i \in \{0,1\}$, where each entry is the optimal $\{\tau, \tau-1\}$-expansion of $\sum_{i=0}^{9} b_i \tau^i$ allowing a maximum exponent of 6 for $\tau - 1$. At most 3 terms of the form $\pm\tau^a(\tau-1)^b$ are required for each representation, so each entry in the table consists of three tuples of the form (d_n, i_n, j_n) representing $d_n \tau^{i_n}(\tau-1)^{j_n}$. Hence, each entry requires 27 bits and the whole look-up table requires 27 KB RAM. Note that, according to the data in Sec. 3.2, using a block size of 5 would still give us a significant improvement over τ-NAF and in this case the table would require less than 1 KB.

Because integer operations are slower than the $\mathbb{F}_{2^m}$ operations in the FAP, the conversion unit will be the bottleneck if the two units use the same clock. So a dual-port RAM is used in order to separate these units into different clock domains. The look-up results are written into the dual-port RAM using one port, and the ECC processor will read them out from the another port later.

4.4 Results

The FPGA design was written in VHDL and implemented on a Xilinx Virtex-II XC2V2000-6. The design was synthesized with Synplify Pro 8.0 and Xilinx ISE 6.2 was used for the place & route. The design (excluding the converter) requires 6,494 slices and 6 BlockRAMs on the XC2V2000-6, and it operates at a maximum clock frequency of $f_{\max} = 128$ MHz. The converter requires 2251 slices, 2 BlockRAMs and 2 multipliers. The maximum clock frequency is 88 MHz. It takes 335 clock cycles, or 3.81 μs to convert one 163-bit integer.

Average timings of the design are presented in Table 4. The latency L_{kP} is

Table 4. Performance calculations of the FPGA implementation on a Xilinx Virtex-II XC2V2000-6 with different values of j. $H_w(k)$ for $J > 0$ are based on empirical data. The numbers of point additions in the mixed coordinates, in $\mathcal{A}$ and in $\mathcal{LD}$ are denoted as $\mathcal{M}$, $\mathcal{A}$ and $\mathcal{LD}$, respectively.

j	$H_w(k)$	$\mathcal{M}$	$\mathcal{A}$	$\mathcal{LD}$	L_{kP}	Time (μs)
0	54.33	53.33	0	0	5735	44.80
2	39.47	36.47	2	2	4675	36.52
3	36.18	32.18	3	3	**4576**	**35.75**
4	34.74	29.74	4	4	4669	36.48
5	33.42	27.42	5	5	4775	37.30
6	32.22	25.22	6	6	4893	38.23

given by (7), and timings are calculated using $f_{\max}$. The time consumed in the conversion is neglected. Table 4 shows that the best performance is achieved when $J = 3$ which is smaller than estimated in Sec. 3.2, because the latencies of point additions differ. In Sec. 3.2, all point additions were assumed equal.

Numerous publications considering implementation of elliptic curve cryptography on FPGAs have been published, e.g., in [24–27]. To the best of the authors' knowledge, the only FPGA-based implementation using τ-NAF expansions was presented by Lutz and Hasan [27] where a kP operation on $E_1(\mathbb{F}_{2^{163}})$ was reported to require $75\,\mu$s on a Xilinx Virtex-E XCV2000E.

5 Further Work

Our results demonstrate that $\{\tau, \tau - 1\}$-expansions lead to a competitive point multiplication algorithm for Koblitz curves when the base point P is not fixed. Nevertheless, there are a number of aspects we are continuing to explore.

The latency of a point multiplication using our FPGA implementation could be significantly reduced at the expense of larger area requirements by computing each row in parallel. This possibility will be studied in the future. In addition, alternative choices of the bases, or even using three bases, may lead to further improvements. For example, using $\{\tau, \tau + 1\}$ may be advantageous in our implementation because computing $(\tau + 1)^j P$ from $(\tau + 1)^{j-1} P$ requires a point addition as opposed to a point subtraction when using bases τ and $\tau - 1$, and subtraction is slightly more expensive than addition.

Our point multiplication algorithm does not require any precomputations involving the base point P nor storage of additional points, and hence is well-suited to applications where P is random. We are investigating the possibility of generalizing window methods, using two-dimensional windows, to our algorithm in order to obtain further improvements when precomputations involving P are permitted.

Although our numerical data suggests that the density of the $\{\tau, \tau - 1\}$-expansions obtained by our conversion algorithm is sublinear in the bit length of k, we do not yet have a proof of this fact. In addition, our conversion algorithm requires a modest amount of storage. These precomputed quantities are independent of both the base point P and multiplier k and can be viewed as part of the domain parameters. Nevertheless, we continue to search for an efficient memory-free conversion algorithm.

References

1. Dimitrov, V., Järvinen, K., Jacobson, Jr., M., Chan, W., Huang, Z.: FPGA implementation of point multiplication on Koblitz curves using Kleinian integers. To appear in CHES 2006 (2006)
2. Koblitz, N.: Elliptic curve cryptosystems. Math. Comp. **48** (1987) 203–209
3. Miller, V.: Use of elliptic curves in cryptography. In: CRYPTO '85. Volume 218 of Lecture Notes in Computer Science (LNCS). (1986) 417–426
4. Koblitz, N.: CM-curves with good cryptographic properties. In: CRYPTO '91. Volume 576 of LNCS. (1992) 279–287

5. National Institute of Standards and Technology (NIST): Digital signature standard (DSS). Federal Information Processing Standard, FIPS PUB 186-2 (2000)
6. Solinas, J.: Efficient arithmetic on Koblitz curves. Designs, Codes, and Cryptography **19** (2000) 195–249
7. Avanzi, R., Heuberger, C., Prodinger, H.: Minimality of the Hamming weight of the τ-naf for Koblitz curves and improved combination with point halving. In: SAC 2005. Volume 3897 of LNCS. (2005) 332–344
8. Dimitrov, V., Jullien, G., Miller, W.: An algorithm for modular exponentiation. Inform. Process. Lett. **66** (1998) 155–159
9. Ciet, M., Sica, F.: An analysis of double base number systems and a sublinear scalar multiplication algorithm. In: Mycrypt 2005. Volume 3715 of LNCS. (2005) 171–182
10. Dimitrov, V., Imbert, L., Mishra, P.: Efficient and secure elliptic curve point multiplication using double-base chains. In: ASIACRYPT 2005. Volume 3788 of LNCS. (2005) 59–78
11. Avanzi, R., Sica, F.: Scalar multiplication on Koblitz curves using double bases. Technical Report Number 2006/067, Cryptology ePrint Archive (2006)
12. Sica, F.: Personal communication. (2006)
13. Conway, J., Smith, D.: On quaternions and octonions. AK Peters (2003)
14. Tijdeman, R.: On integers with many small prime factors. Compos. Math. **26** (1973) 319–330
15. Baker, A.: Linear forms in the logarithms of algebraic numbers IV. Mathematica **15** (1968) 204–216
16. Mignotte, M., Waldshmidt, M.: Linear forms in two logarithms and Schneider's method III. In: Annales Fas. Sci. Toulouse. (1990) 43–75
17. Tijdeman, R.: Personal communication. (2006)
18. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in $GF(2^n)$. In: SAC '98. Volume 1556 of LNCS. (1998) 201–212
19. Doche, C., Lange, T.: Arithmetic of elliptic curves. In Cohen, H., Frey, G., eds.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. Chapman & Hall/CRC (2006) 267–302
20. Higuchi, A., Takagi, N.: A fast addition algorithm for elliptic curve arithmetic in $GF(2^n)$ using projective coordinates. Inform. Process. Lett. **76** (2000) 101–103
21. Al-Daoud, E., Mahmod, R., Rushdan, M., Kilicman, A.: A new addition formula for elliptic curves over $GF(2^n)$. IEEE Trans. Comput. **51** (2002) 972–975
22. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Inform. Comput. **78** (1988) 171–177
23. Wang, C., Troung, T., Shao, H., Deutsch, L., Omura, J., Reed, I.: VLSI architectures for computing multiplications and inverses in $GF(2^m)$. IEEE Trans. Comput. **34** (1985) 709–717
24. Bednara, M., Daldrup, M., von zur Gathen, J., Shokrollahi, J., Teich, J.: Reconfigurable implementation of elliptic curve crypto algorithms. In: IPDPS 2002. (2002) 157–164
25. Leong, P., Leung, K.: A microcoded elliptic curve processor using FPGA technology. IEEE Trans. VLSI Syst. **10** (2002) 550–559
26. Eberle, H., Gura, N., Shantz, S., Gupta, V.: A cryptographic processor for arbitrary elliptic curves over $GF(2^m)$. Technical Report SMLI TR-2003-123, Sun Microsystems, Inc. (2003)
27. Lutz, J., Hasan, A.: High performance FPGA based elliptic curve cryptographic co-processor. In: Proc. of the Int'l Conf. on Information Technology: Coding and Computing. Volume 2. (2004) 486–492