Vassil S. Dimitrov, Kimmo U. Järvinen, Michael J. Jacobson, Jr., Wai Fong (Andy) Chan and Zhun Huang, Provably Sublinear Point Multiplication on Koblitz Curves and Its Hardware Implementation, IEEE Transactions on Computers, vol. 57, no. 11, Nov. 2008, pp. 1469-1481.

Provably Sublinear Point Multiplication on Koblitz Curves and Its Hardware Implementation

Vassil S. Dimitrov, Kimmo U. Järvinen, *Student Member*, *IEEE*,
Michael J. Jacobson Jr., *Member*, *IEEE*, Wai Fong (Andy) Chan, and Zhun Huang

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 $k = \sum \pm\tau^a(\tau-\mu)^b(\tau^2-\mu\tau-1)^c$. 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 nonadjacent form expansions. We present details of an innovative FPGA implementation of our algorithm and performance data demonstrating the efficiency of our method. We also show that implementations with very low computation latency are possible with the proposed method because parallel processing can be exploited efficiently.

Index Terms—Elliptic curve cryptography, field-programmable gate arrays, Koblitz curves, multiple-base expansions, parallel processing, sublinearity.

1 Introduction

In 1985, Koblitz [1] and Miller [2] 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 [3], 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 [4].

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. 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 [3], [5]. Solinas [5] showed how the nonadjacent form (NAF) and window-NAF methods 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 result of Avanzi et al. [6] 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.

Double-base integer representations have been used to devise efficient point multiplication algorithms [7], [8], [9]. 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, which is based on a preliminary version that appeared in CHES 2006 [10], 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 [4]. 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

- *V.S. Dimitrov is with the Department of Electrical and Computer Engineering, University of Calgary, 2500 University Drive NW, Calgary, AB T2N 1N4, Canada. E-mail: dimitrov@atips.ca.*
- *K.U. Järvinen is with the Department of Signal Processing and Acoustics, Helsinki University of Technology, Otakaari 5A, FIN-02150 Espoo, Finland. E-mail: kimmo.jarvinen@tkk.fi.*
- *M.J. Jacobson Jr. and W.F. Chan are with the Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, AB T2N 1N4, Canada. E-mail: {jacobs, chanwf}@cpsc.ucalgary.ca.*
- *Z. Huang is with VIA Technologies, VIA Building, Tsinghua Science Park, No. 1 Zhongguancun East Road, Haidian District, Beijing, China. E-mail: ZhunHuang@viatech.com.cn.*

Manuscript received 31 Oct. 2007; revised 28 Feb. 2008; accepted 13 Mar. 2008; published online 4 Apr. 2008.
Recommended for acceptance by R. Steinwandt, W. Geiselmann, and Ç.K. Koç. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-10-0542.
Digital Object Identifier no. 10.1109/TC.2008.65.

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. This implementation takes advantage of the naturally arising parallelism in our point multiplication algorithm in order to further reduce the latency.

We conjecture that the average density of our representations is sublinear in $\log k$ and provide 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, our numerical experiments indicate that the number of point additions required is within a small constant multiple of $\log k/(\log\log k)$. This constant depends on the size of the precomputed table and was between 3 and 1, with smaller values occurring for larger table sizes. In addition, we provide a proof that sublinearity is obtained using similar expressions involving three bases of the form $\pm\tau^a(\tau-\mu)^b(\tau^2-\mu\tau-1)^c$. Although this provably sublinear algorithm is not as well suited for practical applications, this work is nevertheless of interest, as it represents the first rigorously proven sublinear point multiplication algorithm using complex bases.

Avanzi and Sica [11] have independently investigated the application of 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 the proof has been shown to have a gap [12].

The remainder of this paper is organized as follows: In Section 2, we present our provably sublinear point multiplication algorithm. We present a similar algorithm using only two complex bases in Section 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 Section 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 Section 4. We show in Section 5 how parallel processing can be used for reducing computation latency and demonstrate the efficiency of parallelization with our FPGA implementation. Finally, we conclude with an outlook on possible directions for further research.

2 Multidimensional Frobenius Expansions

We start with the following definitions:

Definition 1. *A complex number ξ of the form $e+f\tau$, where $e,f\in\mathbb{Z}$, 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 $\omega=\pm\tau^x(\tau-\mu)^y(\tau^2-\mu\tau-1)^z$, $x,y,z\geq 0$ is called a $\{\tau,\tau-\mu,\tau^2-\mu\tau-1\}$-*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 that will speed up the computations. The improvements obtained in this paper are based on the following representation, which we will call 2D or 3D Frobenius expansions (or $\{\tau,\tau-1\}$-expansion and $\{\tau,\tau-\mu,\tau^2-\mu\tau-1\}$-expansion, for short):

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

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

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

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

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

Algorithm 1 computes kP given a $\{\tau,\tau-1\}$-expansion of k. In order to simplify, we denote the terms corresponding to $(\tau-1)^l$ in (4) with $r_l(k)$, i.e., $r_l(k)=\sum_{i=1}^{\max(a_{i,l})} s_{i,l}\tau^{a_{i,l}}$. The corresponding algorithm for $\{\tau,\tau-\mu,\tau^2-\mu\tau-1\}$-expansions will be described later, along with a proof that the number of point additions is sublinear in $\log k$. Essentially, kP is computed via a succession of 1D τ-adic expansions.

Algorithm 1 Point multiplication using $\{\tau,\tau-1\}$-expansions.
INPUT: k, P
OUTPUT: $Q=kP$
1: $P_0\leftarrow P$
2: $Q\leftarrow\mathcal{O}$
3: **for** $l=0$ to $\max(b_i)$ **do**
4: $\quad S\leftarrow r_l(k)P_l$ {1D τ-NAF corresponding to $(\tau-1)^l$ in (4)}
5: $\quad P_{l+1}\leftarrow\tau P_l-P_l$
6: $\quad Q\leftarrow Q+S$
7: **end for**

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 $\tau - \mu$ (and $\tau - 1$) cost one Frobenius mapping (free in our computational model) and one point addition or subtraction. The multiplications by $\tau^2 - \mu\tau - 1$ cost two Frobenius mappings and two point additions/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 $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-expansions. The smallest possible value of $\max(b_i)$ and $\max(c_i)$, 0, corresponds to the classical (1D) 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 Algorithm 2. A greedy algorithm for computing $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-expansions is an easy generalization of this algorithm.

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
1: $i \leftarrow 1$
2: **while** $\xi \neq 0$ **do**
3: Find $\omega_i = \pm\tau^{a_i}(\tau - 1)^{b_i}$, $a_i, b_i \geq 0$, the closest $\{\tau, \tau - 1\}$-Kleinian integer to ξ.
4: $\xi \leftarrow \xi - \omega_i$
5: $i \leftarrow i + 1$
6: **end while**

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. 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 Section 2.1, 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 $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-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 nonnegative 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 $\{p_1, p_2, \ldots, p_s\}$-integers if p_s is fixed. The proof depends on the main result of Baker [15] from the theory of linear form in logarithms.

Theorem 3. *Let $a_1, a_2, \ldots, a_k$ be nonzero algebraic integers and $b_1, b_2, \ldots, b_k$ rational integers. Assume $a_1^{b_1} a_2^{b_2} \cdots a_k^{b_k} \neq 1$ and $B = \max(b_1, b_2, \ldots, b_k)$. 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 the 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 Algorithm 2) have 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 $N(\xi) = (a + b\tau)(a + b\overline{\tau})$ is the norm of ξ.*

A 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 gap is due to a reasonable but unproven assumption about the uniform distribution of the arguments of the complex numbers of the form $\tau^a 3^b$ [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 $O(\log N(\zeta)/\ (\log\log N(\zeta)))$ $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-Kleinian integers, such that the largest power of both $\tau - \mu$ and $\tau^2 - \mu\tau - 1$ 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 parts 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

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

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 $2 = \tau(\mu - \tau)$ and $3 = 1 + \mu\tau - \tau^2$, we substitute the 2's and 3's in the {2, 3}-expansions of c_i and d_i to obtain $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-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 $\pm\tau^x(\tau - \mu)^y(\tau^2 - \mu\tau - 1)^z$ 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 $O(\log^\alpha N(\zeta)/(\log\log N(\zeta)))$ $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-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 $\tau - \mu$ and $\tau^2 - \mu\tau - 1$ 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 $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-expansions.

INPUT: A Kleinian integer ζ, a point P on a Koblitz curve, a real constant α with $0 < \alpha < 1/2$

OUTPUT: $Q = \zeta P$

1: Compute in succession for $i = 0, 1, \ldots, \lceil \log^\alpha N(\zeta) \rceil$ the points $P_i^{(1)} = (\tau - \mu)P_{i-1}^{(1)}$ and $P_i^{(2)} = (\tau^2 - \mu\tau - 1)P_{i-1}^{(2)}$, where $P_0^{(1)} = P_0^{(2)} = P$.
2: Compute the points $Q_{i_1,i_2} = P_{i_1}^{(1)} + P_{i_2}^{(2)}$ for $i_i, i_2 = 0, 1, \ldots, \lceil \log^\alpha N(\zeta) \rceil$.
3: Compute a $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-expansion of the form (5) using Theorem 4.
4: Apply in succession τ-NAF based point multiplications based on (5) to compute Q.

The analysis of Algorithm 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 Algorithm 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 $\zeta \equiv k (\mathrm{mod}(\tau^m - 1)/(\tau - 1))$ and applying Algorithm 3 to compute ζP.

Note that the first two steps of Algorithm 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 $\{\tau, \tau - \mu, \tau^2 - \mu\tau - 1\}$-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 $O(\log N(\xi)/\log\log N(\xi))$ $\{\tau, \tau - 1\}$-Kleinian integers such that the largest power of $\tau - 1$ is $O(\log N(\xi)/\log\log N(\xi))$.*

Proof. We shall assume that $b = 0$; otherwise, one applies the same proof for the real and imaginary parts 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 $O(\log N(\xi)/\log\log N(\xi))$. Every block of digits

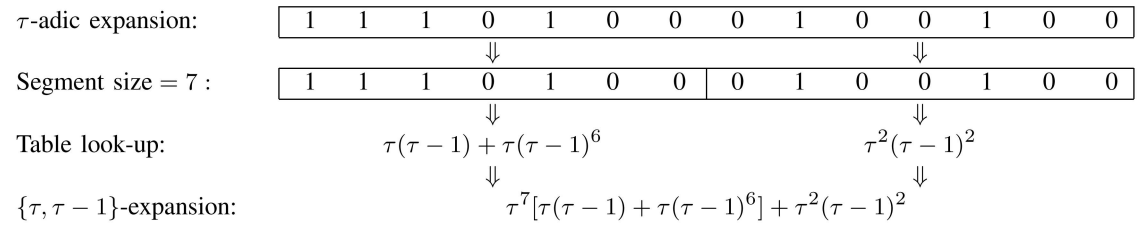

Fig. 1. Expansion of $-104 + 50\tau$ using Algorithm 4.

corresponds to a Kleinian integer and, if Conjecture 1 is true, then every block can be represented as the sum at most

$$O\left(\frac{\log N(\xi)/\log\log N(\xi)}{\log(\log N(\xi)/\log\log N(\xi))}\right) = O\left(\frac{\log N(\xi)}{\log^2\log N(\xi) - \log\log N(\xi)\log\log\log N(\xi)}\right) = O\left(\frac{\log N(\xi)}{\log^2\log N(\xi)}\right)$$

$\{\tau, \tau-1\}$-Kleinian integers. As the number of blocks is $\log\log N(\xi)$, we conclude that this representation consists of at most $O(\log N(\xi)/\log\log N(\xi))$ $\{\tau, \tau-1\}$-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 $O(\log N(\xi)/\log\log N(\xi))$. □

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

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

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

There are four important points regarding the implementation of Algorithm 4:

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 $d_{w-1} \ldots d_1 d_0$ 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 needs to 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 6,465 into a $\{\tau, \tau-1\}$-expansion by using the two different algorithms we have described. Assume that we intend to compute $(6{,}465)P$ for some point $P \in E_1(\mathbb{F}_{2^{17}})$, so $\tau = (1 + \sqrt{-7})/2$. As in the case of computing the τ-NAF expansion, we first do a partial reduction of 6,465 modulo $(\tau^{17} - 1)/(\tau - 1)$ as in [5], yielding $\xi = -104 + 50\tau$. The greedy algorithm, Algorithm 2, returns

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

The blocking algorithm, Algorithm 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 2D 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 maximum power of τ and $(\tau-1)^6$ is the

TABLE 1
The 2D Pictorial Representation of the $\{\tau, \tau-1\}$-Expansion of 6,465 Obtained by Both Algorithms 2 and 4

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

TABLE 2
Average Time (in μs) 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

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 1D τ-adic expansion, and that these are very sparse.

3.2 Numerical Evidence

In this section, we present results from software implementations of Algorithms 2 and 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 [5] of k were implemented in C, using the GMP library for multiprecision 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 [4]. We used the blocking algorithm, Algorithm 4, with $w = 10$ and $\max(b_i) = 6$ to compute the $(\tau, \tau-1)$-expansions, and 500,000 random values of k for each size. Our implementation of Algorithm 4 is faster than that for computing the τ-NAF, by a factor of up to 20 percent.

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, 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 nonzero terms in the expansion plus $\max(b_i) - 1$, as a function of $\max(b_i)$. These average values were computed using 1,000 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 Section 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 [4] when using τ-NAF, our greedy $\{\tau, \tau-1\}$-expansion algorithm (Algorithm 2), and our blocking-based Algorithm 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 500,000 random values of k. Our algorithm requires significantly fewer point additions than τ-NAF in all cases. In addition, the number of point additions required is within a small constant multiple of $\log k/(\log\log k)$, for some constant depending on w, providing evidence in support of Conjecture 1. It is interesting to note that this constant appears to be small, between 3 and 1 according to our data, with smaller values occurring for larger table

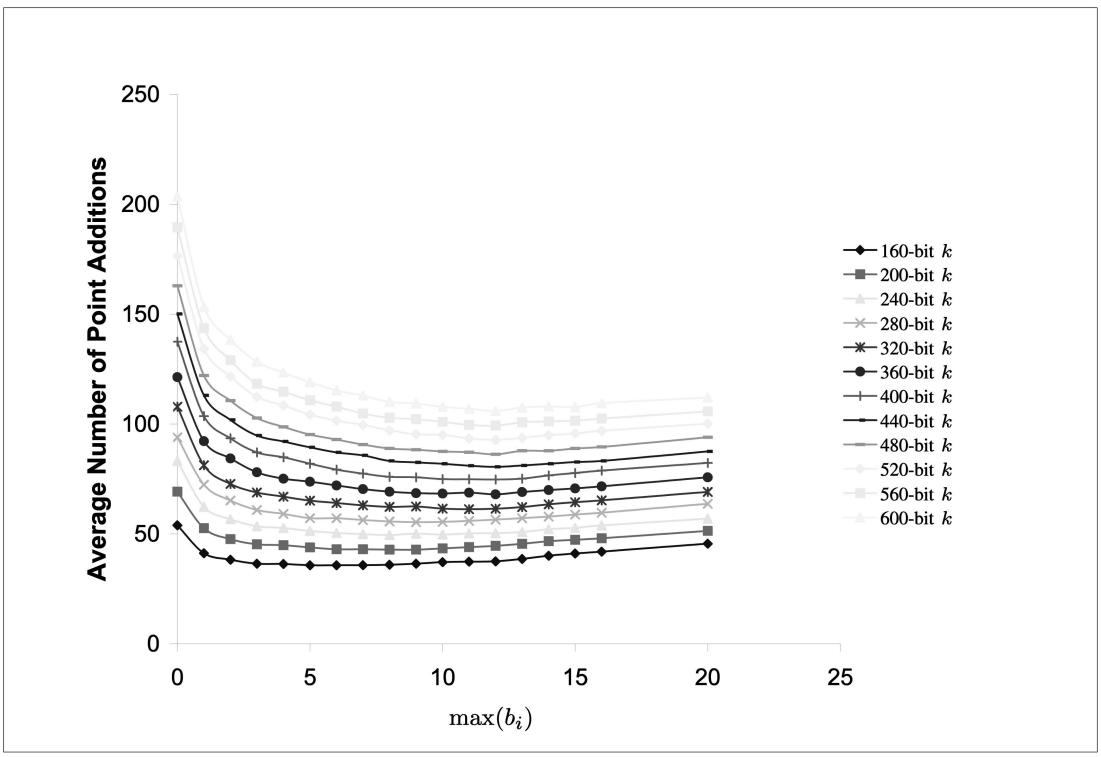

Fig. 2. Average number of point additions to compute kP as a function of $\max(b_i)$ with different bit lengths of k.

TABLE 3
The Average Number of Point Additions Required to Compute kP for the Five Koblitz Curves in [4]

$\log_2 k$	τ-NAF	Alg. 2 (greedy)	Alg. 4 (blocking)		
			$w = 5$	$w = 10$	$w = 16$
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

sizes. Unfortunately, a proof of this observation is currently beyond our reach.

4 Hardware 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 $\mathbb{F}_{2^{163}}$ [4].

As the number of zero coefficients in a $\{\tau, \tau - 1\}$-expansion is large, a normal basis was selected for $\mathbb{F}_{2^m}$. 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 is a cyclic right shift of the bit vector, i.e., $A^2 = (a_{m-1}, a_0, a_1, \ldots, a_{m-2})$, 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 is represented with two coordinates as (x, y) and, in $\mathcal{LD}$, with three coordinates as (X, Y, Z). The $\mathcal{LD}$ triple represents the point $(X/Z, Y/Z^2)$ in $\mathcal{A}$ [18]. When $P = (x, y)$ in $\mathcal{A}$, $-P = (x, x + y)$. Point addition in $\mathcal{A}$ is performed as presented, e.g., in [19], and it costs $\mathrm{I} + 2\mathrm{M} + \mathrm{S} + 8\mathrm{A}$, where I, M, S, and A denote inversion, multiplication, squaring, and addition in $\mathbb{F}_{2^m}$, respectively. Point addition in $\mathcal{LD}$ is performed as presented in [20], and it requires $13\mathrm{M} + 4\mathrm{S} + 9\mathrm{A}$. A mixed coordinate point addition $Q + P$, where Q is in $\mathcal{LD}$ and P in $\mathcal{A}$, requires only $9\mathrm{M} + 5\mathrm{S} + 9\mathrm{A}$ [21]. This cost reduces by M on Koblitz curves and by A if both P and $-P$ are available. Because our implementation stores also $-P$, the cost is $8\mathrm{M} + 5\mathrm{S} + 8\mathrm{A}$. The cost would reduce by one further A if $a = 0$. Point subtraction is simply a point addition with $-P$, i.e., $Q - P = Q + (-P)$. The $\mathcal{A} \mapsto \mathcal{LD}$ mapping does not require any operations in $\mathbb{F}_{2^m}$ while $\mathcal{LD} \mapsto \mathcal{A}$ costs $\mathrm{I} + 2\mathrm{M} + \mathrm{S}$. The cost of a Frobenius mapping is 3S in $\mathcal{LD}$ and 2S in $\mathcal{A}$. An inversion in $\mathbb{F}_{2^m}$ is computed using the Itoh-Tsujii inversion requiring $m - 1$ 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, $\mathrm{I} = 9\mathrm{M} + 162\mathrm{S}$ when $m = 163$.

Different coordinates are used in Algorithm 1 as follows: Point addition in $\mathcal{A}$ is used in computing P_l so that mixed coordinate point addition can be used in $S \leftarrow S \pm P_l$ computations (row 4 in Algorithm 1). Because the results of row multiplications are in $\mathcal{LD}$, $Q \leftarrow Q + S$ is computed in $\mathcal{LD}$.

The design in [10] was implemented in Xilinx Virtex-II, but here, we implemented the design in Altera Stratix II FPGAs [23] so that Altera Stratix II EP2S180 DSP Development Board [24] could be used for prototyping. 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 Section 4.1 and the control logic in Section 4.2. Section 4.3 discusses the conversion unit, i.e., the implementation of the blocking algorithm, Algorithm 4.

4.1 Field Arithmetic Processor (FAP)

The FAP includes a multiplier, a squarer, an adder, a storage element, and a control logic. A storage element for 163-bit elements of $\mathbb{F}_{2^{163}}$ is required in order to store points and temporary variables during computation of kP. As Stratix II offers embedded memory blocks that can be used without consuming logic resources, the storage element is implemented in M4K RAM blocks. One dual-port RAM can be configured into a 256×18-bit mode in M4K. All 163 bits of an element must be accessed in parallel in the FAP architecture. Hence, $\lceil \frac{163}{18} \rceil = 10$ M4Ks are required. Write and read operations require both one clock cycle, i.e., $\mathrm{W} = \mathrm{R} = 1$.

The squarer is a shifter that performs A^{2^d}, where $A \in \mathbb{F}_{2^{163}}$ and $0 \leq d \leq d_{\max} = 2^5 - 1$. Thus, A^{2^d} operations can be performed with a cost of S. We denote these consecutive squaring by S^*. The ability to perform successive squarings in one clock cycle decreases the latencies of inversions and consecutive Frobenius maps significantly. For example, the cost of an inversions reduces from $9\mathrm{M} + 162\mathrm{S}$ to $9\mathrm{M} + 14\mathrm{S}^*$. Addition in $\mathbb{F}_{2^{163}}$ is simply a bitwise exclusive-OR (XOR). Both addition and (successive) squarings are performed in one clock cycle, i.e., $\mathrm{A} = \mathrm{S} = \mathrm{S}^* = 1$.

Field multiplication is critical for the overall performance. The multiplier is a digit-serial implementation of the Massey-Omura multiplier [25]. 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 [25].

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 the maximum 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 $\mathrm{M} = \lceil \frac{163}{24} \rceil + 2 = 9$.

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. Coding is started from the most significant 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 mapping associated with the start bit of the next symbol). If the run of consecutive Frobenius

mappings is longer than $d_{\max}$, the run must be divided into two symbols and $s = 0$, for the latter one. 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 $e \geq H_w k + \max(b_i)$, with equality if and only if $d_{\max} \geq Z(k)$.

The 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 consist of successive FAP instructions directly controlling the FAP. There are control sequences for $P_{l+1} \leftarrow \tau P_l - P_l$ (Frobenius mapping and point addition in $\mathcal{A}$), point addition and subtraction (mixed coordinate point addition), Frobenius mapping, row change (point addition in $\mathcal{LD}$), and $\mathcal{LD} \mapsto \mathcal{A}$ mapping. They are all stored in a ROM implemented in an M4K block.

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 would be the latency of the operation (M, S, or A) plus two clock cycles (R + W). The no-change read mode of Xilinx BlockRAMs was used in [10] for reducing the latency of control sequences. Stratix II devices do not support such read mode, but the latency can be reduced also with read-during-write mode supported by Stratix II memory blocks [23]. The reduction is, however, slightly smaller. When the control sequences were carefully hand-optimized, different operations have the following latencies in clock cycles: the mixed coordinate point addition, $\mathrm{L}_{\mathcal{M}} = 112$, the Frobenius mapping, $\mathrm{L}_F = 7$, row change (point addition in $\mathcal{LD}$), $\mathrm{L}_{\mathcal{LD}} = 167$, the computation of P_l, $\mathrm{L}_{P_l} = 171$, and the $\mathcal{LD} \mapsto \mathcal{A}$ mapping, $\mathrm{L}_{\mathcal{LD} \mapsto \mathcal{A}} = 145$. The first point addition of each row is simply $S \leftarrow \pm P_l$ and the first row change operation is given by $Q \leftarrow S$. Both of these operations have a latency of $\mathrm{L}_C = 6$. In the beginning, an initialization is performed including, e.g., the transferring of P into the FAP. The latency of the initialization is $\mathrm{L}_I = 11$. Thus, it follows that the latency of the kP computation becomes

$$\begin{aligned}\mathrm{L}_{kP} = &(H_w(k) - (\max(b_i)+1))\mathrm{L}_{\mathcal{M}} + (\max(b_i)+2)\mathrm{L}_C + \mathrm{L}_I \\ &+ (e - \max(b_i))\mathrm{L}_F + \max(b_i)(L_{\mathcal{LD}} + \mathrm{L}_{P_l}) + \mathrm{L}_{\mathcal{LD} \mapsto \mathcal{A}},\end{aligned} \tag{6}$$

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

$$\mathrm{L}_{kP} = 119\, H_w(k) + 232\, \max(b_i) + 56. \tag{7}$$

4.3 Conversion Unit

The conversion unit, which converts an integer k into a $\{\tau, \tau - 1\}$-expansion, is a straightforward implementation of Algorithm 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 $\times$ 18-bit embedded multiplier to create 102-bit $\times$ 102-bit products. The ALU also includes adders, shifters, and the rounding function required by the partial reduction algorithm [5]. 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.

TABLE 4
Performance Calculations of the FPGA Implementation on an Altera Stratix II EP2S180F1020C3 with Different Values of $\max(b_i)$ When the Clock Frequency Is 152 MHz

$\max(b_i)$	$H_w(k)$	$\mathcal{M}$	$\mathcal{A}$	$\mathcal{LD}$	L_{kP}	Time (μs)
0	54.33	53.33	0	0	6522	42.91
2	41.27	38.27	2	2	5431	35.73
3	38.44	34.44	3	3	**5326**	**35.04**
4	37.09	32.09	4	4	5398	35.51
5	36.04	30.04	5	5	5504	36.21
6	34.96	27.96	6	6	5609	36.90

$H_w(k)$ for $\max(b_i) > 0$ are averages from 10,000 random 163-bit integers converted with Algorithm 4 when $w = 10$. 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.

For our implementation, we selected the block size $w = 10$, so every 10 bits of the τ-adic expansion are used as an index into a lookup 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 three 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 lookup table requires a 27-Kbit RAM. Note that, according to the data in Section 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 Kbit.

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 dual-port RAMs are used in order to separate these units into different clock domains. The lookup results are written into the dual-port RAMs using one port, and the FAP will read them out from the other port later.

4.4 Results

The design was written in VHDL and implemented in an Altera Stratix II EP2S180F1020C3 FPGA [23], which is on the Altera Stratix II EP2S180 DSP Development Board. The design was synthesized with Quartus II 6.0 SP1. In total, the design requires 8,799 Adaptive Logic Modules (ALMs) and 20 M512 and 36 M4K memory blocks including logic for interfacing and buffers separating clock domains. The FAP and its control logic require 6,084 ALMs and 16 M4Ks, and they operate at the maximum clock frequency of 152.28 MHz. The converter requires 1,947 ALMs, 8 M4Ks, and 4 DSP blocks and operates at 93.54 MHz. It takes 335 clock cycles, or 3.58 μs, to convert one 163-bit integer.

Average timings of the design are presented in Table 4. The latency L_{kP} is given by (7), and timings are calculated using a clock of 152 MHz. The time consumed in the conversion is neglected because the $\{\tau, \tau - 1\}$-expansion for the next kP can be computed simultaneously while the FAP processes the previous kP. Table 4 shows that the best performance is achieved when $\max(b_i) = 3$, which is smaller than estimated in Section 3.2, because the latencies of point additions differ. In Section 3.2, all point additions were assumed equal.

The implementation on Xilinx Virtex-II computes kP in 35.75 μs [10], which shows that the longer latency caused by the lack of the no-change read mode in Stratix II is compensated by a higher clock frequency and the designs run slightly faster on Stratix II. Numerous publications considering implementation of elliptic curve cryptography on FPGAs have been published, e.g., in [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], and [38]. To the authors' knowledge, the only published FPGA-based implementations using τ-NAF expansions were presented in [32], [34], and [35], and they all use the NIST K-163 curve, $E_1(\mathbb{F}_{2^{163}})$. Okada et al. [35] published an implementation that computes a kP operation in 45.6 ms with an Altera EPF10K250AGC599-2, and Lutz and Hasan [34] presented a design that computes kP in 75 μs on a Xilinx Virtex-E XCV2000E. Järvinen et al. [32] presented an implementation using parallel processing blocks that can compute up to 166,000 three-term point multiplications per second with an average computation time of 114.2 μs on a Stratix II S180C3 FPGA. Comparing designs implemented on different FPGAs is difficult because it is hard to tell which portion of a performance difference is caused by different implementation platforms and which by some novel implementation technique. Anyhow, the design we have presented is faster than any of the above-mentioned designs. Moreover, it will be shown in Section 5 how our method can be further accelerated with parallel processing.

5 Parallel Implementation

This section presents an efficient method for reducing the latency of kP by using parallel processing. It is well known that reducing the latency of kP with parallelism is hard because of limited intrinsic parallelism in the operation if it is computed with traditional methods such as the binary double-and-add method. However, when k is a $\{\tau, \tau-1\}$-integer, parallelism can be efficiently used for reducing the latency of our algorithm because rows of the table representing k can be computed in parallel.

Parallel computation is performed as follows: The implementation consists of p identical processing blocks (FAP and control logic), which can exchange data with each other. Processing blocks compute their subcomputations independently and the results of the subcomputations are combined with $p-1$ point additions resulting in kP. The critical path consists of the most expensive subcomputation and $\lceil \log_2 p \rceil$ point additions because parallelism can be utilized also in combining the subcomputations.

Let the area of a single processing block be A_0. Because identical parallel processing blocks are used, the area of p processing blocks is approximately[1] pA_0. Area efficiency of parallel implementations is measured via the latency-area product $R = CA$, where C is the critical path and A is the area. The best area efficiency is obtained with a setup that minimizes R.

The maximum number of parallel processing blocks is $p_{\max} = \max(b_i) + 1$, i.e., the number of rows. Because the rows corresponding to $(\tau-1)^l$ with small l have a higher average number of nonzero terms than the rows with large l as can be seen in Fig. 3, using $p_{\max}$ parallel processing blocks leads to an inefficient R although C is small. In practice, selecting $p < p_{\max}$ and assigning rows to processing blocks by their computational cost leads to both small C and R.

1. Some of the interface logic can be shared, but some extra logic is required in order to connect the processing blocks.

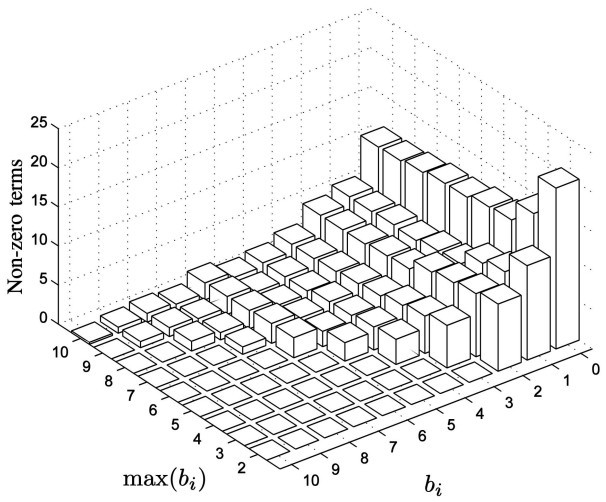

Fig. 3. The average distribution of nonzero terms in 10,000 randomly selected $\{\tau, \tau-1\}$-integers, where $\max(b_i) \in [2, 10]$, obtained by the blocking algorithm with the block size $w = 10$.

When a row is computed, the base point $P_l = (\tau-1)^l P$ needs to be computed first. If the base point is computed with sequential $(\tau-1)P$ computations, in total l point subtractions and Frobenius mappings are required. This is not a problem if $p = 1$ because all rows are computed in the same processing block, but when $p > 1$, one would need to compute also the base points that are not needed in order to get P_l with high l. For example, in order to get $P_4 = (\tau-1)^4 P$, four point subtractions and Frobenius mappings are required, i.e., P_1, P_2, and P_3 need to be computed, although they may not be needed.

The cost of base point computations can be reduced by utilizing the fact that $\tau^2 + \mu\tau + 2 = 0$. For $E_1(\mathbb{F}_{2^m})$, $\mu = -1$ and $\tau^2 + 2 = \tau$ give the following nonsequential formulas for $(\tau-1)^l$, where $2 \le l \le 4$:

$$\begin{aligned}(\tau-1)^2 &= \tau^2 - 2\tau + 1\\ &= \tau^2 + 2 - 2 - 2\tau + 1\\ &= \tau - 2\tau - 1\\ &= -\tau - 1,\end{aligned} \tag{8}$$

$$\begin{aligned}(\tau-1)^3 &= (\tau-1)(-\tau-1)\\ &= -\tau^2 - \tau + \tau + 1\\ &= -\tau^2 + 1,\end{aligned} \tag{9}$$

$$\begin{aligned}(\tau-1)^4 &= (\tau-1)(-\tau^2+1)\\ &= (\tau^2+1)(-\tau^2+1)\\ &= -\tau^4 + \tau^2 - \tau^2 + 1\\ &= -\tau^4 + 1.\end{aligned} \tag{10}$$

$\{\tau, \tau-1\}$-Kleinian integer has the following distribution of non-zero terms and costs:

l	0	1	2	3	4	5	6
$H_w(r_l(k))$	10	5	4	4	2	2	9
Cost	9	5	4	4	2	3	10

Scheduling performs with Alg. 5 as follows:

Row 6 $\rightarrow$ Block 1 (cost 10)
Row 0 $\rightarrow$ Block 2 (cost 9)
Row 1 $\rightarrow$ Block 3 (cost 5)
Row 2 $\rightarrow$ Block 3 (cost $5+4+1=10$)
Row 3 $\rightarrow$ Block 2 (cost $9+4+1=14$)
Row 5 $\rightarrow$ Block 1 (cost $10+3+1=14$)
Row 4 $\rightarrow$ Block 3 (cost $10+2+1=13$)

which results in the following schedule:

Block	l	Cost
1	5, 6	14 (13)
2	0, 3	14
3	1, 2, 4	13

Fig. 4. Scheduling example for $p=3$. The scheduling is performed for a 163-bit integer $k = 1902cd968f45b3c58932fdb63eea875b2884b3d35_x$, which is converted using the blocking algorithm, Algorithm 4, with $w=10$ and $\max(b_i)=6$. The most expensive computation is performed in the second processing block and it has a critical path of 14 point additions. The cost of the first processing block reduces by one because base point computations for P_5 and P_6 can be combined resulting in a reduction of one point subtraction.

Nonsequential formulas including only one point addition or subtraction cannot be found for $l > 4$, but (8)-(10) can be still used for reducing the computational cost. For example,

$$\begin{aligned}(\tau-1)^{11}P &= (\tau-1)^4((\tau-1)^4((\tau-1)^3P)) \\ &= (-\tau^4+1)\big((-\tau^4+1)\big((-\tau^2+1)P\big)\big),\end{aligned}$$

which requires only three point subtractions instead of 11. Generally, $(\tau-1)^l P$ requires $\lceil l/4 \rceil$ point subtractions. Similar equations cannot be found for $\{\tau, \tau-1\}$-Kleinian integers and $E_0(\mathbb{F}_{2^m})$, where $\mu=1$ and $\tau^2+2=-\tau$, but for $\{\tau, \tau+1\}$-Kleinian integers, similar formulas are given as follows:

$$\begin{aligned}(\tau+1)^2 &= \tau-1, \\ (\tau+1)^3 &= \tau^2-1, \\ (\tau+1)^4 &= -\tau^4+1.\end{aligned}$$

Analogously, there are no formulas for $\{\tau, \tau+1\}$-Kleinian integers and $E_1(\mathbb{F}_{2^m})$.

The assignment of rows is performed as presented in Algorithm 5, which produces a computation schedule including the indices of the rows. Cost of the row l is $H_w(r_l(k)) - 1 + \lceil l/4 \rceil$, where $H_w(r_l(k))$ is the number of nonzero terms in the row. An example of scheduling is presented in Fig. 4. The example shows that, in that particular case, the critical path can be reduced from $36-1+6=41$ to $14-1+\lceil \log_2 3 \rceil = 15$ point additions by using three parallel processing blocks instead of one.

Algorithm 5 Greedy scheduling algorithm
INPUT: Costs of rows, $\max(b_i)$, p
OUTPUT: Computation schedule
1: **for** $i=0$ to $\max(b_i)$ **do**
2: Find the processing block with the smallest assigned cost (if more than one, select the one with the smallest index)
3: Find the nonassigned row with the highest cost (if more than one, select the one with the smallest index)
4: Assign the row to the processing block
5: Add the cost of the row to the assigned cost of the block and remove the row from the list of nonassigned rows
6: If the assigned row is not the first row computed in the block, add the cost of combining rows to the assigned cost of the block
7: **end for**

It is not obvious which are the optimal choices for $\max(b_i)$ and p when C and R are both considered. Hence, Algorithm 5 was performed for 10,000 random 163-bit integers converted with Algorithm 4 $(w=10)$ and different $\max(b_i)$ and p. The resulted critical paths and latency-area products are shown in Table 5. The critical path C is the number of point additions. All point additions are assumed to have a cost of one, reflecting the situation where only $\mathcal{A}$ is used for representing points on $E_a(\mathbb{F}_{2^m})$. The best possible critical paths and latency-area products with different p (bold values in Table 5) are plotted in Fig. 5, which clearly shows that the critical path shortens considerably when $p > 1$. However, major reductions are not achievable with $p > 4$ because the most expensive row becomes a bottleneck. Latency-area product always increases when p grows, but the increase is moderate when $p \leq 4$. The most expensive row bounds the critical path, and as can be seen in Fig. 3 in Section 5, the cost of the most expensive row decreases very slowly when $\max(b_i)$ grows. Thus, the technique yields efficient results only with small p. This is not a major concern in practice as large p are often unreachable anyhow because of area constraints.

5.1 Hardware Implementation

The efficiency of the parallelization technique was studied in practice by designing an FPGA implementation based on the design described in Section 4. Because the design uses both $\mathcal{A}$ and $\mathcal{LD}$, the analysis of optimal p and $\max(b_i)$ performed in Section 5 is not valid.

The design computes kP so that, when the integer is converted with the converter described in Section 4.3, the scheduling algorithm, Algorithm 5, is performed for the $\{\tau, \tau-1\}$-expansion. It was implemented so that it assumes that point additions in $\mathcal{A}$ and $\mathcal{LD}$ cost 1.5 times as much as a mixed coordinate point addition. The correct values are 1.53 and 1.49, respectively, so the error of this assumption is negligible, but the implementation is easier and the result more area efficient. The scheduling increases the latency of the conversion by 16 clock cycles resulting in 351 clock cycles. After the conversions, the subcomputations are performed independently in parallel FAPs. The computations are

TABLE 5
Critical Paths and Latency-Area Products with $p \in [1,6]$ and $\max(b_i) \in [2,10]$

p	1		2		3		4		5		6	
$\max(b_i)$	C	R	C	R	C	R	C	R	C	R	C	R
2	42.27	42.27	24.13	48.26	21.85	65.55						
3	40.44	40.44	22.13	44.27	17.91	53.73	16.98	67.91				
4	40.09	40.09	**21.58**	**43.15**	16.47	49.40	14.86	59.44	15.72	78.59		
5	40.04	40.04	21.75	43.49	**16.34**	**49.03**	14.41	57.63	15.13	75.64	15.11	90.65
6	**39.96**	**39.96**	22.13	44.26	16.37	49.10	**13.94**	**55.77**	14.47	72.33	14.41	86.45
7	40.57	40.57	22.81	45.63	16.69	50.07	14.01	56.04	14.34	71.72	14.25	85.53
8	41.04	41.04	23.52	47.05	17.08	51.24	13.95	55.80	14.03	70.16	13.87	83.22
9	41.63	41.63	24.78	49.56	17.89	53.68	14.25	56.98	**13.93**	**69.64**	13.59	81.55
10	42.51	42.51	26.25	52.50	18.82	56.45	14.83	59.32	14.09	70.45	**13.58**	**81.46**

Critical path C is given as the number of point additions including cost of combining subcomputations, and latency-area product is $R = pC$. The best values for all p are in bold.

performed as discussed in Section 4 with the exception that P_i are computed using (8)-(10). In the end, the results of the subcomputations are combined with point additions in $\mathcal{A}$, each requiring 171 clock cycles.

Latencies for the above-described design with different p and $\max(b_i)$ are presented in Table 6. Fig. 6 depicts the shortest latencies and the smallest latency-area products for all p. Comparisons of Tables 5 and 6 and Figs. 5 and 6 show that the parallelization technique gives similar results than in Section 5 also for our FPGA design. Also, in this case, $p = 4$ is the highest number of parallel processing blocks that result in any significant reductions in latency. However, the values of $\max(b_i)$ that result in the shortest latencies differ from the analysis of Section 5. The speedups are also slightly smaller than in Section 5. Based on Table 6, we selected the parameters $p = 4$ and $\max(b_i) = 6$ for the FPGA implementation.

5.2 Results

The parallel implementation was also written in VHDL and synthesized for Stratix II S180C3 with Quartus II. The design requires 28,328 ALMs, which is 39 percent of the device resources, 52 M512s, and 66 M4Ks. The design operates with 152- and 93-MHz clocks similarly as the one processor implementation that was expected because the critical path determining the maximum clock frequency is the same in both cases. The average kP latency is 2,033 clock cycles as in Table 6. Thus, a kP computation requires on average only 13.38 μs. Conversions including schedulings require 3.77 μs.

The speedup compared to the single processing block implementation presented in Section 4.4 is $35.04/13.38 = 2.61$ (excluding the conversion) or $38.62/17.15 = 2.25$ (including the conversion). The increase in area is $28{,}328/8{,}799 = 3.21$ (including the converter). This shows that very high speedups are achievable with moderate area increase by using parallel processing. Furthermore, the resulted implementation computing kP on average in 17.15 μs (including the conversion) is the fastest published FPGA-based implementation, at least to the authors' knowledge.

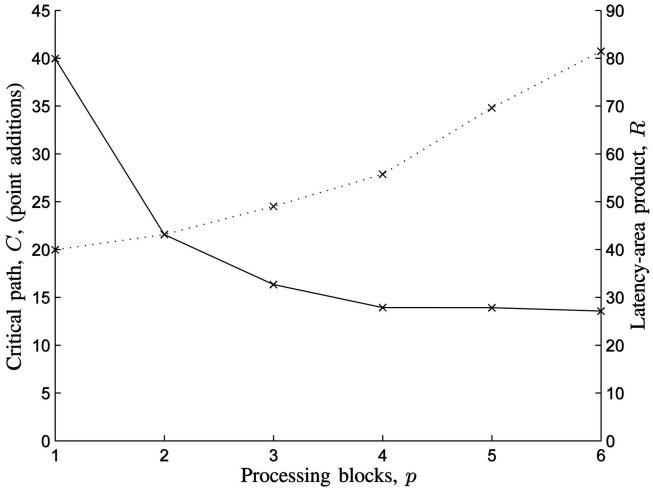

Fig. 5. The (solid line) best critical paths C and (dotted line) latency-area products R, i.e., the bold values in Table 5, with $p \in [1,6]$.

6 Conclusions and Further Work

Our results have demonstrated 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.

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 $(\tau+1)^l P$ from $(\tau+1)^{l-1} P$ 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. $\{\tau, \tau+1\}$-expansions may also be useful in parallel implementations because $(\tau+1)^l P$ can be computed efficiently on $E_0(\mathbb{F}_{2^m})$ but $(\tau-1)^l P$ cannot.

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 2D windows, to our algorithm in order to obtain further improvements when precomputations involving P are permitted.

TABLE 6
Critical Path and Latency-Area Product Estimates with $p \in [1,6]$ and $\max(b_i) \in [2,10]$ for the FPGA Implementation

p	1		2		3		4		5		6	
$\max(b_i)$	C	R	C	R	C	R	C	R	C	R	C	R
2	5431	5431	3202	6405	2885	8656						
3	**5326**	**5326**	**3008**	**6016**	2483	7450	2311	9246				
4	5401	5401	3021	6043	**2355**	**7065**	2085	8341	2221	11103		
5	5519	5519	3098	6196	2392	7176	2056	8223	2153	10765	2143	12858
6	5639	5639	3227	6453	2460	7379	**2033**	**8131**	2086	10430	2061	12366
7	5868	5868	3329	6657	2511	7533	2072	8289	2079	10396	2045	12268
8	6078	6078	3447	6895	2578	7733	2102	8409	**2059**	**10295**	2004	12026
9	6322	6322	3590	7180	2673	8018	2151	8606	2065	10325	1979	11874
10	6626	6626	3698	7396	2733	8199	2183	8733	2083	10414	**1976**	**11857**

Critical path is given as the number of clock cycles including cost of combining subcomputations, and latency-area product is $R = pC$. The best values for all p are in bold.

It is also possible to augment our conversion algorithm using a sliding window analog, in which we slide the blocks along the τ-adic expansion of the scalar so that the low-order bit is always one. Experiments indicate that this does not significantly reduce the required number of point additions, but it does have the advantage that the size of the lookup table is cut in half.

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.

ACKNOWLEDGMENTS

Vassil S. Dimitrov and Michael J. Jacobson Jr. were supported in part by NSERC of Canada.

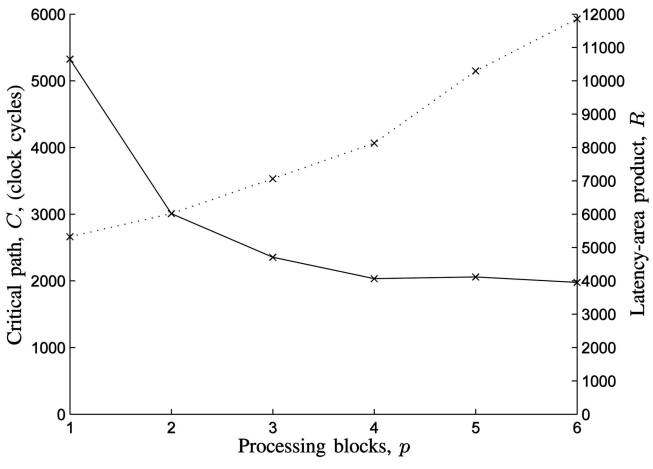

Fig. 6. The (solid line) best critical path C and (dotted line) latency-area product R estimates for the FPGA implementation, i.e., the bold values in Table 6, with $p \in [1,6]$.

REFERENCES

[1] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Computation,* vol. 48, pp. 203-209, 1987.
[2] V. Miller, "Use of Elliptic Curves in Cryptography," *Advances in Cryptology—CRYPTO '85,* pp. 417-426, 1986.
[3] N. Koblitz, "CM-Curves with Good Cryptographic Properties," *Advances in Cryptology—CRYPTO '91,* pp. 279-287, 1992.
[4] *Digital Signature Standard (DSS),* Fed. Information Processing Standard, FIPS PUB 186-2, Nat'l Inst. of Standards and Technology (NIST) Computer Security FIPS PUB 186-2, Jan. 2000.
[5] J. Solinas, "Efficient Arithmetic on Koblitz Curves," *Designs, Codes and Cryptography,* vol. 19, pp. 195-249, 2000.
[6] R. Avanzi, C. Heuberger, and H. Prodinger, "Minimality of the Hamming Weight of the τ-NAF for Koblitz Curves and Improved Combination with Point Halving," *Selected Areas in Cryptography—SAC '05,* pp. 332-344, 2005.
[7] V. Dimitrov, G. Jullien, and W. Miller, "An Algorithm for Modular Exponentiation," *Information Processing Letters,* vol. 66, no. 3, pp. 155-159, 1998.
[8] M. Ciet and F. Sica, "An Analysis of Double Base Number Systems and a Sublinear Scalar Multiplication Algorithm," *Progress in Cryptology—Mycrypt '05,* pp. 171-182, 2005.
[9] V. Dimitrov, L. Imbert, and P. Mishra, "Efficient and Secure Elliptic Curve Point Multiplication Using Double-Base Chains," *Advances in Cryptology—ASIACRYPT '05,* pp. 59-78, 2005.
[10] V.S. Dimitrov, K.U. Järvinen, M.J. Jacobson Jr., W.F. Chan, and Z. Huang, "FPGA Implementation of Point Multiplication on Koblitz Curves Using Kleinian Integers," *Cryptographic Hardware and Embedded Systems—CHES '06,* pp. 445-459, 2006.
[11] R. Avanzi and F. Sica, "Scalar Multiplication on Koblitz Curves Using Double Bases," *Progress in Cryptology—VIETCRYPT '06,* pp. 131-146, 2006.
[12] F. Sica, *Scalar Multiplication on Koblitz Curves Using Double Bases.* Univ. of Calgary, invited talk, Apr. 2006.
[13] J. Conway and D. Smith, *On Quaternions and Octonions.* AK Peters, 2003.
[14] R. Tijdeman, "On Integers with Many Small Prime Factors," *Composition Math.,* vol. 26, no. 3, pp. 319-330, 1973.
[15] A. Baker, "Linear Forms in the Logarithms of Algebraic Numbers IV," *Math.,* vol. 15, pp. 204-216, 1968.
[16] M. Mignotte and M. Waldshmidt, "Linear Forms in Two Logarithms and Schneider's Method III," *Annales de la Faculté des Sciences de Toulouse,* pp. 43-75, 1990.
[17] R. Tijdeman, personal communication, 2006.
[18] J. López and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$," *Selected Areas in Cryptography—SAC '98,* pp. 201-212, 1998.
[19] C. Doche and T. Lange, "Arithmetic of Elliptic Curves," *Handbook of Elliptic and Hyperelliptic Curve Cryptography,* Chapman and Hall/CRC, H. Cohen and G. Frey, eds., chapter 13, pp. 267-302, 2006.
[20] A. Higuchi and N. Takagi, "A Fast Addition Algorithm for Elliptic Curve Arithmetic in $GF(2^n)$ Using Projective Coordinates," *Information Processing Letters,* vol. 76, no. 3, pp. 101-103, 2000.

[21] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman, "A New Addition Formula for Elliptic Curves over $GF(2^n)$," *IEEE Trans. Computers,* vol. 51, no. 8, pp. 972-975, Aug. 2002.
[22] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases," *Information and Computation,* vol. 78, no. 3, pp. 171-177, Sept. 1988.
[23] *Stratix II Device Handbook,* Altera, http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf, May 2007.
[24] *Stratix II EP2S180 DSP Development Board—Reference Manual,* Altera, http://www.altera.com/literature/manual/mnl_SII_DSP_RM_11Aug06.pdf, Aug. 2006.
[25] C. Wang, T. Troung, H. Shao, L. Deutsch, J. Omura, and I. Reed, "VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$," *IEEE Trans. Computers,* vol. 34, no. 8, pp. 709-717, Aug. 1985.
[26] B. Ansari and M.A. Hasan, "High Performance Architecture of Elliptic Curve Scalar Multiplication," Technical Report CACR 2006-1, Univ. of Waterloo, 2006.
[27] S. Bajracharya, C. Shu, K. Gaj, and T. El-Ghazawi, "Implementation of Elliptic Curve Cryptosystems over $GF(2^n)$ in Optimal Normal Basis on a Reconfigurable Computer," *Proc. Int'l Conf. Field Programmable Logic and Application (FPL '04),* pp. 1098-1100, 2004.
[28] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms," *Proc. Reconfigurable Architectures Workshop, Int'l Parallel and Distributed Processing Symp. (IPDPS '02),* pp. 157-164, Apr. 2002.
[29] M. Benaissa and W. Lim, "Design of Flexible $GF(2^m)$ Elliptic Curve Cryptography Processors," *IEEE Trans. Very Large Scale Integration Systems,* vol. 14, no. 6, pp. 659-662, June 2006.
[30] R. Cheung, N. Telle, W. Luk, and P. Cheung, "Customizable Elliptic Curve Cryptosystem," *IEEE Trans. Very Large Scale Integration Systems,* vol. 13, pp. 1048-1059, Sept. 2005.
[31] H. Eberle, N. Gura, S. Shantz, and V. Gupta, "A Cryptographic Processor for Arbitrary Elliptic Curves over $GF(2^m)$," Technical Report SMLI TR-2003-123, Sun Microsystems, May 2003.
[32] K. Järvinen, J. Forsten, and J. Skyttä, "FPGA Design of Self-Certified Signature Verification on Koblitz Curves," *Cryptographic Hardware and Embedded Systems—CHES '07,* pp. 256-271, 2007.
[33] P. Leong and K. Leung, "A Microcoded Elliptic Curve Processor Using FPGA Technology," *IEEE Trans. Very Large Scale Integration Systems,* vol. 10, no. 5, pp. 550-559, Oct. 2002.
[34] J. Lutz and A. Hasan, "High Performance FPGA Based Elliptic Curve Cryptographic Co-Processor," *Proc. Int'l Conf. Information Technology: Coding and Computing (ITCC '04),* vol. 2, pp. 486-492, Apr. 2004.
[35] S. Okada, N. Torii, K. Itoh, and M. Takenaka, "Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA," *Cryptographic Hardware and Embedded Systems—CHES '00,* pp. 25-40, 2000.
[36] G. Orlando and C. Paar, "A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$," *Cryptographic Hardware and Embedded Systems—CHES '00,* pp. 41-56, 2000.
[37] F. Rodríguez-Henríquez, N. Saqib, and A. Díaz-Pérez, "A Fast Parallel Implementation of Elliptic Curve Point Multiplication over $GF(2^m)$," *Microprocessors and Microsystems,* vol. 28, nos. 5-6, pp. 329-339, Aug. 2004.
[38] C. Shu, K. Gaj, and T. El-Ghazawi, "Low Latency Elliptic Curve Cryptography Accelerators for NIST Curves over Binary Fields," *Proc. IEEE Int'l Conf. Field-Programmable Technology (FPT '05),* pp. 309-310, Dec. 2005.

Vassil S. Dimitrov received the PhD degree in applied mathematics from the Bulgarian Academy of Sciences in 1995. Since 1995, he has held postdoctoral positions at the University of Windsor, Ontario (1996-1998) and Helsinki University of Technology (1999-2000). During the period 1998-1999, he worked as a research scientist for Cigital, Dulles, Virginia (formerly known as Reliable Software Technology), where he conducted research work on different cryptanalysis problems. Since 2001, he has been an associate professor in the Department of Electrical and Computer Engineering, University of Calgary, Alberta. His main research areas include implementation of cryptographic protocols, number-theoretic algorithms, computational complexity, image processing and compression, and related topics.

Kimmo U. Järvinen received the MSc (Tech) degree in electrical engineering in 2003 from Helsinki University of Technology (TKK), where he is currently working toward the DSc (Tech) degree. He has been with the Department of Signal Processing and Acoustics (formerly Signal Processing Laboratory) at TKK since 2002 and in the Graduate School in Electronics, Telecommunications and Automation (GETA) since 2004. In autumn 2005, he was on a research visit at the University of Calgary. His research interests include FPGA implementations of cryptographic algorithms, especially elliptic curve cryptography. He is a student member of the IEEE.

Michael J. Jacobson Jr. received the BSc (Hon) and MSc degrees from the University of Manitoba and the Dr rer nat degree from the Technical University of Darmstadt in 1999. He is an associate professor in the Department of Computer Science, University of Calgary, and a member of the Centre for Information Security and Cryptography. His research interests are cryptography and related applications of computational number theory, especially as applied to algebraic number fields and function fields. He is a member of the IEEE and the ACM.

Wai Fong (Andy) Chan is currently working toward the master's degree in computer science in the Department of Computer Science, University of Calgary, Alberta. He is also working for Symantec on antivirus software projects. His previous industrial experience includes 10 years as a computer engineer for Samsung.

Zhun Huang received the PhD degree in electronics from Tsinghua University, Beijing, in 2005. He was with the Department of Electrical and Computer Engineering, University of Calgary, Canada, from 2005 to 2006. In 2007, he joined VIA Technologies, Beijing. His research interests include cryptographic VLSI, random number generator, and computer arithmetic.