Microprocessors and Microsystems xxx (2008) xxx–xxx

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Fast point multiplication on Koblitz curves: Parallelization method and implementations

Kimmo Järvinen *, Jorma Skyttä

Helsinki University of Technology, Department of Signal Processing and Acoustics, Otakaari 5A, FIN-02150 Espoo, Finland

ARTICLE INFO

Keywords:
Elliptic curve cryptography
Field programmable gate array
Koblitz curves
Parallelism

ABSTRACT

Point multiplication is required in every elliptic curve cryptosystem and its efficient implementation is essential. Koblitz curves are a family of curves defined over $\mathbb{F}_{2^m}$ allowing notably faster computation. We discuss implementation of point multiplication on Koblitz curves with parallel field multipliers. We present a novel parallelization method utilizing point operation interleaving. FPGA implementations are described showing the practical feasibility of our method. They compute point multiplications on average in 4.9 µs, 8.1 µs, and 12.1 µs on the standardized curves NIST K-163, K-233, and K-283, respectively, in an Altera Stratix II FPGA.

1. Introduction

Koblitz [1] and Miller [2] independently suggested using elliptic curves in public-key cryptography in 1985. Since then elliptic curve cryptography has attained considerable amount of interest in the cryptographic research community, and increasingly also in the industry. The main reason for the attractiveness of elliptic curves is that shorter keys can be used for attaining similar level of security than in traditional public-key cryptography schemes based on the difficulty of integer factorization or discrete logarithm. For example, elliptic curve cryptography achieves approximately the same level of security with 173 bits than RSA with 1024 bits [3]. Studies have shown that elliptic curve cryptography is superior to RSA also in terms of speed and area required in implementation [4–6].

Field programmable gate arrays (FPGAs) offer many advantages in implementing cryptographic algorithms because they provide fast performance and flexibility [7]. Hence, numerous studies on FPGA implementation of elliptic curve cryptography have been published including [6,8–20]. A comprehensive survey of the field is presented in [21].

The principal operation required in every elliptic curve cryptosystem is called point multiplication. Much effort has been allocated in developing methods for its efficient computation because it acts as the bottleneck in elliptic curve cryptosystems. A comprehensive review can be found in [22], for example. Point multiplication is commonly known to be an operation which is hard to parallelize because of data dependencies and much of the research has concentrated on studying effects of parallelization. Finite field multiplication dominates in the cost of point multiplication and, thus, reducing the number of field multiplications on the critical path is essential. The fastest method for computing point multiplication without precomputations, called Montgomery point multiplication, was presented by López and Dahab in [23]. Cheung et al. [13] and Rodríguez-Henríquez et al. [18] showed that Montgomery point multiplication can be efficiently computed with four parallel field multipliers.

In 1991, Koblitz [24] suggested using a special family of elliptic curves nowadays referred to as Koblitz curves. Point multiplication is considerably more efficient on them than on general curves. Koblitz curves have been widely studied in the academia and they have been included also in certain standards, such as [25–27]. Koblitz curves have attained some interest also in the FPGA community as they have been considered in [9,14–17].

The problem that we are addressing is that computations on Koblitz curves cannot be parallelized as efficiently as Montgomery point multiplication. We recently presented a study on parallelization of Koblitz curve computations in [14] but the methods presented are not as effective as those available for Montgomery point multiplication. Only three parallel field multipliers can be utilized in Koblitz curve point multiplication with the existing methods [14] whereas Montgomery point multiplication can efficiently use up to four multipliers [18,13]. Hence, the more multipliers are available the smaller is the benefit of using Koblitz curves. Koblitz curves are faster than general curves even if parallel field multipliers are available, but the difference becomes smaller which makes Koblitz curves less attractive.

The contributions of our paper are twofold:

(1) We present a simple and efficient method for speeding up Koblitz curve computations when parallel field multipliers

* Corresponding author. Tel.: +358 9 451 5176; fax: +358 9 452 3614.
E-mail addresses: kimmo.jarvinen@tkk.fi (K. Järvinen), jorma.skytta@tkk.fi (J. Skyttä).

0141-9331/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2008.08.002

are available. The method is based on point operation interleaving and it achieves full field multiplier utilization with two or four field multipliers. Montgomery point multiplication can be implemented efficiently with two or four multipliers as well [18,13]. Hence, both Koblitz curves and general curves can achieve their full potential in the same hardware. Our method implies that Koblitz curves are approximately three times faster than general curves. Furthermore, our method can be applied also in point multiplication algorithms requiring precomputations whereas methods presented in [18,13] cannot which increases the difference even further.

(2) We describe a highly optimized architecture based on our method which achieves very fast point multiplication times. The feasibility of the method and the architecture is demonstrated by providing several implementations on an Altera Stratix II FPGA. The implementations achieve average point multiplication times of only 4.9 µs, 8.1 µs, and 12.1 µs on K-163, K-233, and K-283, respectively, which are curves recommended by National Institute of Standards and Technology (NIST) [25]. We also study the effects of field basis selection and conclude that polynomial basis provides faster results than normal basis.

The remainder of the paper is organized as follows. Section 2 presents the preliminaries of elliptic curve cryptography and discusses computation of point multiplication and Koblitz curves. We describe our method in Section 3. Implementations showing the practical feasibility of the method are presented in Section 4 and implementation results are given and compared to other published results in Section 5. We end with conclusions and suggestions for future research in Section 6.

2. Preliminaries

Elliptic curves defined over finite binary fields, denoted by $\mathbb{F}_{2^m}$, are commonly used in practical cryptosystems. These curves are called binary curves. We consider binary curves of the following form:

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

where $a, b \in \mathbb{F}_{2^m}$ with $b \neq 0$. Henceforth, curves of this form are called general curves.

Let $E(\mathbb{F}_{2^m})$ denote the set of points on E. A point (x, y) is in $E(\mathbb{F}_{2^m})$ if it satisfies Eq. (1). Also a point called the point at infinity, $\mathcal{O}$, is a point in $E(\mathbb{F}_{2^m})$. Points in $E(\mathbb{F}_{2^m})$ form an additive abelian group with $\mathcal{O}$ as an identity element. The additive operation of the group is referred to as point addition and it is defined as $P_3 = P_1 + P_2$ where $P_i \in E(\mathbb{F}_{2^m})$.

Point multiplication, which is a basic component of every elliptic curve cryptosystem, is defined by using point additions as follows:

$$Q = kP = \underbrace{P + P + \cdots + P}_{k \text{ times}} \tag{2}$$

where $Q, P \in E(\mathbb{F}_{2^m})$ and k is an integer. P is called the base point and Q is the result point. Security of elliptic curve cryptosystems is based on the difficulty of solving the inverse operation of point multiplication called elliptic curve discrete logarithm problem (ECDLP), i.e., the problem of finding k if P and Q are given.

Point multiplication decomposes into three levels of hierarchy from top to bottom as follows:

- Point multiplication,
- Point operations, and
- Finite field arithmetic.

These hierarchy levels are discussed in the following sections by concentrating on the subjects that are most relevant for this paper. Our contributions target to the two highest levels of the hierarchy, and they are considered in detail in Sections 2.1 and 2.2. The lowest level is also considered shortly in Section 2.3. Finally, Koblitz curves are discussed in Section 2.4.

2.1. Point multiplication

Point doubling is a special case of point addition, $P_3 = P_1 + P_2$, where $P_1 = P_2$ and, henceforth, point addition refers solely to the operation $P_3 = P_1 + P_2$ where $P_1 \neq P_2$. Point additions and point doublings can be used in computing Eq. (2) when the integer k is represented with binary expansion as

$$k = \sum_{i=0}^{\ell-1} k_i 2^i, \quad \text{where} \quad k_i \in \{0, 1\}. \tag{3}$$

The simplest point multiplication algorithm is called double-and-add algorithm and it scans the bits of k in order starting either from the least significant bit (lsb) or from the most significant bit (msb). Point doubling is performed for every bit, but point addition is required only if $k_i = 1$. The length of k is $\ell \approx m$ and, thus, the double-and-add algorithm requires on average m point doublings and $m/2$ point additions.

Because point addition is not needed if $k_i = 0$, it is of interest to reduce the number of nonzeros. A simple, but effective, option is to use a signed-bit representation, i.e., $k_i \in \{0, \pm 1\}$, called non-adjacent form (NAF). NAF has the property that adjacent bits are never both nonzeros, i.e., $k_i k_{i+1} = 0$ for all i. Every k has a unique NAF and it has the minimum number of nonzeros among all signed-bit representations. Let $H(k)$ denote the Hamming weight of k, i.e., the number of nonzeros in k. When k is in NAF, $H(k) \approx m/3$. NAF is especially useful in point multiplication because point subtraction, $P_3 = P_1 - P_2 = P_1 + (-P_2)$, has roughly the same cost as point addition. When integers are in NAF, a modification of the double-and-add algorithm is used called double-and-add-or-subtract algorithm. Otherwise it is similar to the double-and-add algorithm, but point subtraction is computed when $k_i = -1$. Thus, the average cost of point multiplication is m point doublings and $m/3$ point additions or point subtractions.

Further reductions in computational requirements can be achieved by allowing precomputations involving the base point P. Such methods include window methods and combings, for example, but they are not considered in depth in this paper. However, the proposed method can be used also for such methods as will be discussed in Section 3.

Montgomery's ladder [28] is a point multiplication algorithm which performs both point addition and point doubling in every iteration of the algorithm. Hence, it has the cost of m point doublings and m point additions. The efficiency of Montgomery's ladder arises from the fact that point multiplication can be computed without information of the y-coordinate. Thus, the main loop of the algorithm operates only on the x-coordinate and the y-coordinate of the result point is retrieved in the end. This leads to very efficient point addition and point doubling. An adaptation of Montgomery's idea for binary curves was presented by López and Dahab in [23] and, henceforth, Montgomery point multiplication refers to their algorithm.

2.2. Point operations

The traditional point representation with two coordinates as (x, y) is referred to as the affine coordinate representation, or $\mathcal{A}$ for short. If $P = (x, y)$, point negation is given by $-P = (x, x + y)$. Point doubling, point addition, and point subtraction all require an inversion in $\mathbb{F}_{2^m}$. The exact costs are $I + 2M + S + 6A$, $I + 2M + S + 8A$, and $I + 2M + S + 9A$, respectively, where I, M, S, and A stand for inversion, multiplication, squaring, and addition in $\mathbb{F}_{2^m}$. Inversions are expensive and, thus, it is of interest to trade inversions to other operations in $\mathbb{F}_{2^m}$.

Inversions can be avoided by using projective coordinates where a point is represented with three coordinates as (X, Y, Z). Several types of projective coordinates have been introduced (see [29], for example) and we consider López-Dahab coordinates [30], or $\mathcal{LD}$ for short, where a point (X, Y, Z) represents the point $(X/Z, Y/Z^2)$ in $\mathcal{A}$. The $\mathcal{LD}$ coordinates allow an efficient mixed coordinate point addition (and point subtraction). If $P_1 = (X_1, Y_1, Z_1)$ is in $\mathcal{LD}$ and $P_2 = (x_2, y_2)$ is in $\mathcal{A}$, point addition $P_3 = (X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (x_2, y_2)$ is given as follows [31]

$$\begin{aligned} &A = Y_1 + y_2 Z_1^2; \quad B = X_1 + x_2 Z_1; \quad C = BZ_1; \quad Z_3 = C^2; \quad D = x_2 Z_3 \\ &X_3 = A^2 + C(A + B^2 + aC); \quad Y_3 = (D + X_3)(AC + Z_3) + (y_2 + x_2)Z_3^2 \end{aligned} \tag{4}$$

which require $9M + 5S + 9A$.

The traditional projective coordinates, where a point (X, Y, Z) represents the affine point $(X/Z, Y/Z)$, are another interesting coordinate system, because they are used in Montgomery point multiplication. Both point addition and point doubling are required in every iteration and together they cost $6M + 4S + 3A$ [23].

2.3. Finite field arithmetic

The lowest level of the hierarchy composes of arithmetic operations in a finite field. We consider two different representations of binary fields; namely, polynomial basis and normal basis.

A binary field $\mathbb{F}_{2^m}$ with polynomial basis is constructed by representing elements as binary polynomials of degree at most $m - 1$, i.e., $a(x) = \sum_{i=0}^{m-1} a_i x^i$ where $a_i \in \{0,1\}$. In normal basis elements are represented as $A = \sum_{i=0}^{m-1} a_i \beta^{2^i}$ with a basis of the form $\{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$ with the property that $\beta^{2^m} = \beta$. Thus, m bits are required in representing an element in both bases.

Addition is performed in both bases with a bitwise exclusive-or (XOR). Field multiplication is more efficient in polynomial basis where it is computed by multiplying polynomials modulo an irreducible polynomial $p(x)$. Squaring, on the other hand, is cheap in normal basis where it is a simple rotation of the bit vector. Inversion is the most expensive field operation regardless of the basis. Inversions can be computed by using Extended Euclidean Algorithm or its modifications or Fermat's Little Theorem. We use Fermat's Little Theorem as proposed by Itoh and Tsujii in [32]. An Itoh–Tsujii inversion has the cost:

$$I = (\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)M + (m-1)S. \tag{5}$$

2.4. Koblitz curves

Koblitz curves [24] are a family of curves defined by Eq. (1) with $a \in \{0,1\}$ and $b = 1$. We denote a Koblitz curve by E_K. Koblitz curves have the appealing feature that if the point $P = (x, y)$ is on E_K, so is the point (x^2, y^2). This operation is called Frobenius map and we denote it by $\phi(P)$. The inexpensiveness of the Frobenius maps can be exploited efficiently in point multiplication because Frobenius maps can replace point doublings.

It stands for all points in $E_K(\mathbb{F}_{2^m})$ that $\phi^2(P) + 2P = \mu\phi(P)$, where $\mu = (-1)^{1-a}$. Thus, ϕ can be seen as a complex number τ satisfying $\tau^2 + 2 = \mu\tau$ which gives $\tau = (\mu + \sqrt{-7})/2$. Frobenius maps replace point doublings if k is represented with τ-adic expansion as

$$k = \sum_{i=0}^{\ell-1} k_i \tau^i. \tag{6}$$

Solinas presented efficient algorithms for finding τ-adic non-adjacent form (τNAF) in [33]. τNAF is analogous with the NAF discussed above as it has on average the same length and Hamming weight.

An algorithm for point multiplication on E_K with $\mathcal{LD}$ coordinates and k in τNAF is presented in Algorithm 1. On average, Algorithm 1 requires $m - 1$ Frobenius maps and $m/3 - 1$ point additions or point subtractions, because the first point addition or point subtraction is simply a substitution. As Frobenius maps are almost free, point additions, i.e., computations of Eq. (4), define the performance of Algorithm 1. The cost of Eq. (4) reduces on Koblitz curves to $8M + 5S + 9A$ if $a = 1$ or $8M + 5S + 8A$ if $a = 0$, because aC in the computation of X_3 does not require multiplication.

Algorithm 1. Double-and-add-or-subtract algorithm in $\mathcal{LD}$ coordinates for Koblitz curves

```
Input: $P = (x, y)$ on $E_K$, integer $k = \sum_{i=0}^{\ell-1} k_i \tau^i$ where $k_i \in \{0, \pm 1\}$ and $k_{\ell-1} = \pm 1$
Output: $Q = kP$
  $Q \leftarrow (x, y, 1)$ if $k_{\ell-1} = 1$ or $Q \leftarrow (x, x + y, 1)$ if $k_{\ell-1} = -1$
  for $i = \ell - 2$ downto 0 do
    $Q \leftarrow \phi(Q) = (X^2, Y^2, Z^2)$
    if $k_i \neq 0$ then
      $Q \leftarrow Q + k_i P = (X, Y, Z) \pm (x, y)$   /* Computed with Eq. (4) */
    end if
  end for
  $Q \leftarrow (X/Z, Y/Z^2)$
```

3. Description of the method

Our new method exploits parallelism on the two highest levels of the hierarchy (see Section 2) and results in an optimal utilization of field multipliers. The key idea is that although point operation requires the result of the previous point operation, parts of it can be processed with the data that is available before the previous operation is finished. Thus, it is possible to use resources that are free due to the inefficiencies of point operations for computing parts of the next operation. We begin the description of our method by analyzing the second highest level of the hierarchy from which we proceed by showing how these results can be used on the highest level.

Fig. 1 plots the data dependency graph of Eq. (4) and highlights the operations that are needed for computing different coordinates of P_3. We assume that the point (x_2, y_2) is always available, which is the case in Algorithm 1 where it is either P or $-P$. Obviously, the Z_3 computation requiring two multiplications is the simplest operation. It requires only that Z_1 and X_1 are available, i.e., Y_1 is not needed. The two multiplications of the X_3 computation require X_1, Y_1, Z_1, and $C = \sqrt{Z_3}$. Hence, Z_3 must be ready. However, the first multiplication requires only Z_1 and it can be computed before Y_1 and C are available. The Y_3 computation requires four multiplications, three of which can be computed if Z_3 and Y_1 are available but one of them requires also X_3. The above description and Fig. 1 show that the number of multiplications on the critical path is five with two multipliers and four with three or more multipliers, as already pointed out in [14].

Next, we redefine Eq. (4) by using eight subcomputations which all include one multiplication and certain additions and squarings. We name them $\mathcal{Z}_0$, $\mathcal{Z}_1$, $\mathcal{X}_0$, $\mathcal{X}_1$, $\mathcal{Y}_0$, $\mathcal{Y}_1$, $\mathcal{Y}_2$, and $\mathcal{Y}_3$ so that $\mathcal{Z}_0$ includes the first multiplication of the Z_3 computation, $\mathcal{Z}_1$ includes the second one, etc. The subcomputations are as follows:

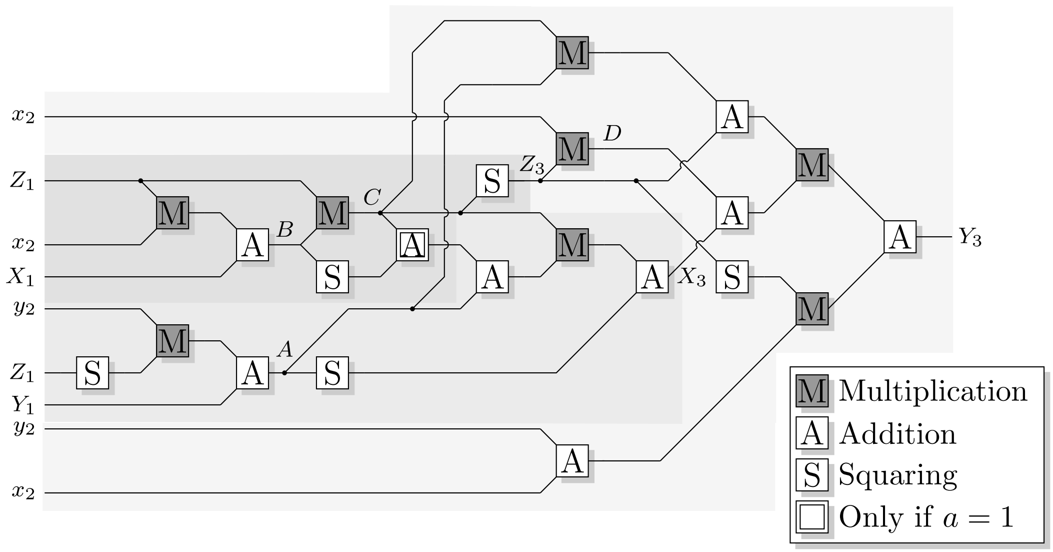

Fig. 1. Data dependency graph of Eq. (4) on Koblitz curves, i.e., $a \in \{0,1\}$. Operations needed for computing X_3, Y_3, and Z_3 are highlighted.

$$\mathcal{Z}_0: \quad E = x_2 Z_1 \tag{7}$$

$$\mathcal{Z}_1: \quad \begin{cases} C = Z_1(E + X_1) \\ F = aC + (E + X_1)^2 \\ Z_3 = C^2 \end{cases} \tag{8}$$

$$\mathcal{X}_0: \quad G = y_2 Z_1^2 \tag{9}$$

$$\mathcal{X}_1: \quad X_3 = C(F + G + Y_1) + (G + Y_1)^2 \tag{10}$$

$$\mathcal{Y}_0: \quad H = C(G + Y_1) + Z_3 \tag{11}$$

$$\mathcal{Y}_1: \quad D = x_2 Z_3 \tag{12}$$

$$\mathcal{Y}_2: \quad J = Z_3^2(x_2 + y_2) \tag{13}$$

$$\mathcal{Y}_3: \quad Y_3 = H(D + X_3) + J \tag{14}$$

The subcomputations compute Eq. (4) when they are performed sequentially from $\mathcal{Z}_0$ to $\mathcal{Y}_3$. When several multipliers are available, Eq. (4) can be computed as follows.

Two multipliers. First, one computes $\mathcal{Z}_0$ by using the first multiplier. Second, the same multiplier computes $\mathcal{Z}_1$ finishing the Z_3 computation and, at the same time, $\mathcal{X}_0$ is computed in the second multiplier. After this, $\mathcal{Y}_0$ and $\mathcal{Y}_2$ are computed in the first multiplier and the second multiplier computes $\mathcal{X}_1$, $\mathcal{Y}_1$, and $\mathcal{Y}_3$. Thus, five multiplications are on the critical path.

Three multipliers. The first multiplier computes $\mathcal{Z}_0$, $\mathcal{Z}_1$, and $\mathcal{Y}_0$. The second multiplier computes $\mathcal{X}_0$, $\mathcal{X}_1$, and $\mathcal{Y}_2$ and it starts computing when the first multiplier has finished $\mathcal{Z}_0$. The third multiplier computes $\mathcal{Y}_1$ and $\mathcal{Y}_3$ after the first multiplier has finished $\mathcal{Z}_1$. The critical path becomes four multiplications.

Four multipliers. Point addition can be computed so that $\mathcal{Z}_0$ and $\mathcal{Z}_1$ are computed in the first multiplier, $\mathcal{X}_0$ and $\mathcal{X}_1$ in the second, $\mathcal{Y}_0$ and $\mathcal{Y}_2$ in the third, and $\mathcal{Y}_1$ and $\mathcal{Y}_3$ in the fourth. The second multiplier begins computations when the first multiplier has finished $\mathcal{Z}_0$ and the third and the fourth multiplier begin when $\mathcal{Z}_1$ is ready. The critical path is again four multiplications. The reason for including a four multiplier scheme although the critical path is not reduced is that it results in a very efficient interleaved computation schedule as will be shown in the following.

The above computation schedules do not change the critical path bounds described in [14]. However, they allow considerable improvements on the highest level. These improvements are possible because Frobenius maps following each point addition can be performed so that each coordinate is mapped as soon as it is ready; i.e., Z_3 is mapped after $\mathcal{Z}_1$, X_3 after $\mathcal{X}_1$ and Y_3 after $\mathcal{Y}_3$. This allows interleaving successive point additions so that all multipliers begin processing the next point addition as soon as they have finished their part of the previous point addition (and Frobenius maps).

We first consider the case of two multipliers. Because $\mathcal{Z}_0$ does not require Y_3, $\mathcal{Z}_0$ can be performed simultaneously with $\mathcal{Y}_3$ of the previous point addition. A computation schedule where successive point additions are interleaved is shown in Fig. 2a. For simplicity, Frobenius maps following point additions are neglected in Fig. 2a. The effective critical path of point addition contains only four multiplications and, hence, the latency of point multiplication excluding additions and squarings becomes

$$(4(H(k) - 1) + 1)M + I. \tag{15}$$

It is impossible to achieve full multiplier utilization with three multipliers, but an effective critical path of three multiplications is still achievable. The first multiplier can compute $\mathcal{Z}_0$ simultaneously while the second and third multiplier process $\mathcal{Y}_2$ and $\mathcal{Y}_3$ as shown in Fig. 2b. The third multiplier is idle approximately one-third of time. The latency of point multiplication is given by

$$(3(H(k) - 1) + 1)M + I. \tag{16}$$

When four multipliers are available, $\mathcal{Z}_0$ of the next point addition can be performed simultaneously with $\mathcal{X}_1$, $\mathcal{Y}_0$, and $\mathcal{Y}_1$. Furthermore, $\mathcal{Z}_1$ and $\mathcal{X}_0$ of the next point addition can be computed in the first and second multiplier while the third and fourth multiplier process $\mathcal{Y}_2$ and $\mathcal{Y}_3$. All multipliers are hence always occupied as can be seen in Fig. 2c. The effective critical path reduces to only two multiplications, and the latency of point multiplication is

$$2H(k)M + I. \tag{17}$$

3.1. Discussion

The method can be used for various point multiplication algorithms on Koblitz curves including algorithms requiring precomputations. The only requirement is that the algorithm uses consecutive mixed coordinate point additions, i.e., Eq. (4). In fact, the latencies given by Eqs. (15)–(17) are valid also for window and combing point multiplication algorithms if the latencies of precomputations are neglected. The method can be used also for

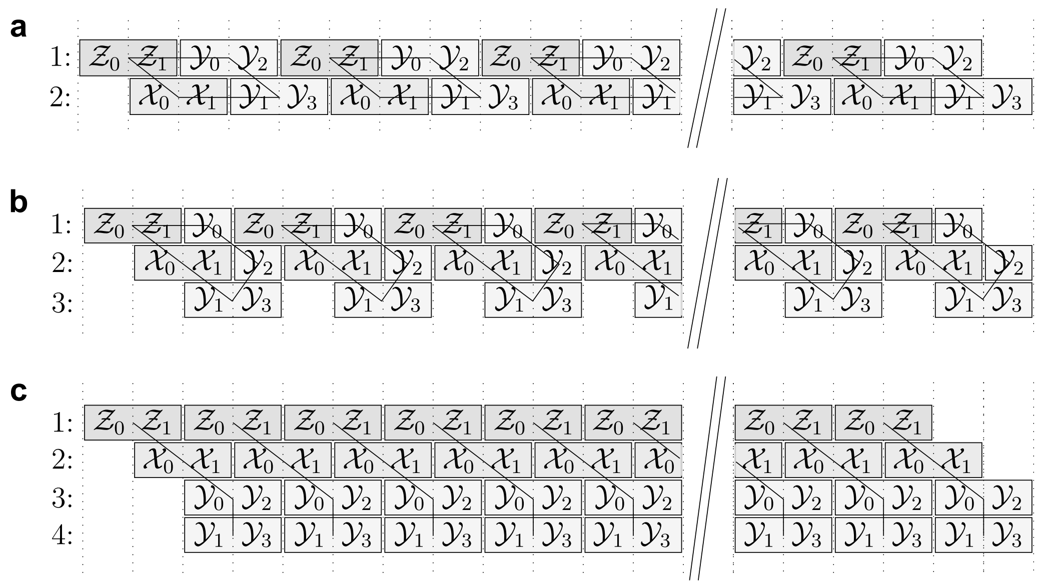

Fig. 2. Computation schedules with (a) two, (b) three, or (c) four multipliers. Operations connected with a line belong to the same point addition.

the double-base approach presented in [9] but the benefits are smaller because also other point operations besides mixed coordinate point additions are on the critical path. Furthermore, our method can be combined with the parallelization method presented in [14] which splits point multiplication on Koblitz curves for several processors yielding further improvements in performance.

Although we considered the use of the method only on Koblitz curves, it can be generalized also for general curves. However, point doublings are then required instead of Frobenius maps. Point doubling in $\mathcal{LD}$ requires five multiplications [34] with a minimum critical path of three multiplications. Operations can be interleaved so that the effective critical path of point doubling is three multiplications with two multipliers and two multiplications with three or more multipliers. Unfortunately, this result has little practical significance because point doublings alone result in a critical path of at least $2m$ multiplications. Montgomery point multiplication computes the entire point multiplication with $2m$ multiplications on the critical path [18,13] and, hence, it is faster. If support for both general and Koblitz curves is needed, then Montgomery point multiplication should be used for general curves and our method for Koblitz curves. They both can be implemented efficiently using the same hardware. Operation interleaving cannot achieve any benefits in Montgomery point multiplication because both X and Z coordinates[1] are needed in the first operation.

The speedup of using Koblitz curves instead of general curves is studied in Table 1 which shows the number of multiplications required in the main loop of Montgomery point multiplication on general curves and double-and-add-or-subtract algorithm on Koblitz curves. Only multiplications in the main loop are compared in Table 1 in order to simplify the analysis. This skews the analysis only little because multiplications dominate in point multiplication and the cost of coordinate conversion is negligible compared to the cost of the main loop.

A direct implication of our method is that point multiplication on Koblitz curves can be computed roughly three times as fast as on general curves with equal amount of hardware. This follows from the fact that both Montgomery point multiplication and our method require two multiplications per iteration. Montgomery point multiplication, however, requires m iterations whereas our method requires on average only $m/3$ iterations with k in τNAF.

Table 1
Comparison of the number of multiplications on the critical path of the main loop in point multiplication on general curves, E, and Koblitz curves, E_K

Multipliers	E [13,18]	E_K [14]	Speedup	E_K (This work)	Speedup
1	6mM	$\frac{8}{3}$mM	2.25	n/a	n/a
2	3mM	$\frac{5}{3}$mM	1.80	$\frac{4}{3}$mM	2.25
3	3mM	$\frac{4}{3}$mM	2.25	mM	3.00
4	2mM	$\frac{4}{3}$mM	1.50	$\frac{2}{3}$mM	3.00

The speedup column indicates how much faster point multiplication on Koblitz curves is compared to general curves with that particular method.

Table 1 also shows that our method is twice as fast as the best existing method for Koblitz curves. The existing methods also suggest that the more multipliers are in use the less benefit is gained from using Koblitz curves. However, our method shows that the case is actually on the contrary: the more multipliers are available the faster Koblitz curves are compared to general curves.

4. Implementations

Several implementations were designed in order to study the practical feasibility of our method. We only consider implementations with four multipliers because we target to the fastest performance. Although the method and the following architecture are not restricted to any particular Koblitz curve, we limit the analysis to the curves K-163, K-233, and K-283 recommended by NIST [25]. The curves are defined by Eq. (1) so that $a = b = 1$ for K-163 and $a = 0$ and $b = 1$ for K-233 and K-283 [25]. Even the smallest field size $m = 163$ was recently shown to provide high security [35].

The architecture is described in the following. Field multipliers are discussed in Section 4.1 and coordinate conversion needed in the end is considered in Section 4.2. Latency formulae for point multiplication are derived in Section 4.3 and discussion on the architecture is given in Section 4.4.

As discussed in Section 2.3, a finite field can be constructed using polynomial or normal basis, and the curves included in [25] are defined over both bases. The advantages of the bases are

[1] The Y coordinate is not computed in the main loop as it is recovered in the end (see Section 2.1).

fast multiplication in polynomial basis and almost free squaring (and Frobenius map) in normal basis. We implemented both normal basis and polynomial basis versions in order to study the effects of basis selection.

In order to minimize latencies caused by other operations besides multiplications and Frobenius maps, operation-specific processing units which compute additions and squarings in parallel with multiplications were designed and optimized specifically for X_3, Y_3 and Z_3 computations.

The processing unit computing Z_3, called the Z processor, is optimized for $\mathcal{Z}_0$ and $\mathcal{Z}_1$ subcomputations, and it is used exclusively for them. The Z processor is built around a multiplier as presented in Fig. 3a. The Z processor computes the Z coordinate of d consecutive Frobenius maps by using a squarer designed for successive squarings (the FR block in Fig. 3a). In normal basis, this squarer is a simple shifter and the shifter can perform up to 31 successive squarings (Frobenius maps) in one clock cycle. In polynomial basis, each squaring requires one clock cycle and, thus, d successive Frobenius maps cost d clock cycles. Implementation of the two squarings also depends on the basis. In normal basis, these squarings are performed by rearranging the bitvectors and no logic is required, but a simple circuitry is needed in polynomial basis (reduction modulo $p(x)$).

The X processor computes $\mathcal{X}_0$ and $\mathcal{X}_1$ and it consists of a multiplier, adders and squarers as shown in Fig. 3b. An additional select signal s is needed because one of the multiplier inputs needs to be squared in $\mathcal{X}_0$ but not in $\mathcal{X}_1$ as shown in Eqs. (9) and (10).

The Y processor computing $\mathcal{Y}_0$–$\mathcal{Y}_3$ uses two multipliers as shown in Fig. 3c and it is the most complex of the three processors. The upper multiplier in Fig. 3c computes multiplications of $\mathcal{Y}_0$ and $\mathcal{Y}_3$ while the lower one is used for $\mathcal{Y}_1$ and $\mathcal{Y}_2$.

Latencies are equal for all processors. In normal basis all outputs are available after $M + 1$ clock cycles. As d Frobenius maps require d clock cycles in polynomial basis, out_{FR} is available after $M + d$ clock cycles whereas other outputs are ready in $M + 1$ clock cycles.

Algorithm 1 is implemented using computation schedule presented in Fig. 2. First, the base point $P = (x, y)$ is loaded into the processor which then computes and stores the y-coordinate of $-P = (x, x + y)$. The registers holding X_3 and Z_3 are initialized to $X_3 = x^d$ and $Z_3 = 1$ where d is the number of Frobenius maps following the msb of k. Initialization of the Y_3 register depends on the msb of k. If $k_{\ell-1} = 1$, $Y_3 = y^d$ and, if $k_{\ell-1} = -1$, $Y_3 = (x + y)^d$ (See Algorithm 1).

The schedule of Fig. 2c consists of two cycles. When the inputs and outputs of the processors are connected as shown in Table 2, the processors compute Algorithm 1 as scheduled in Fig. 2c. In the beginning and in the end, the processors which are idle are not enabled. Because the Z processor starts computing the next point addition immediately after it has finished Frobenius maps but Z_3 is still needed in the X and Y processors, a register is needed for storing Z_3.

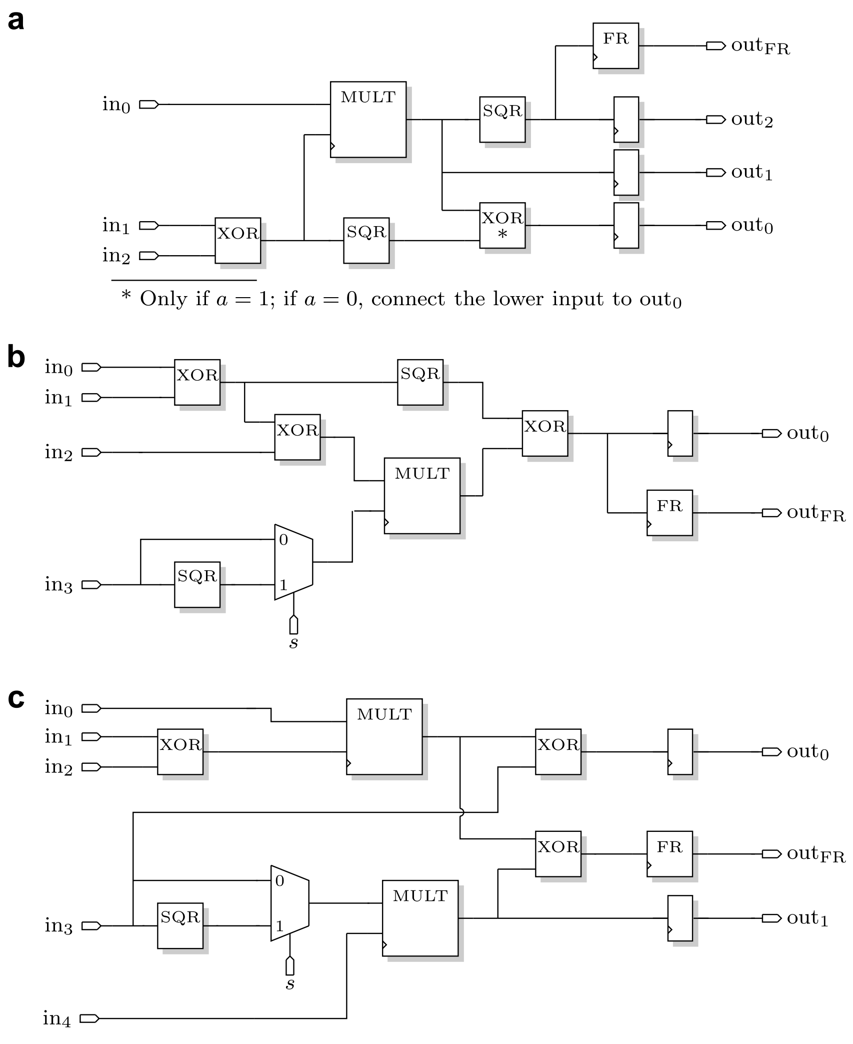

Fig. 3. (a) Z processor, (b) X processor, and (c) Y processor.

Table 2
Inputs and outputs of the processing units

Processor	Input/ouput	1st cycle	2nd cycle
Z	In_0	Z_1	Z_1
	In_1	x	E
	In_2	0	X_1
	Out_0	n/a	F
	Out_1	E	C
	Out_2	n/a	Z_3
	$\mathrm{Out}_{\mathrm{FR}}$	n/a	Z_3^d
X	In_0	Y_1	0
	In_1	G	0
	In_2	F	$y/x + y$
	In_3	C	Z_1
	s	0	1
	Out_0	X_3	G
	$\mathrm{Out}_{\mathrm{FR}}$	X_3^d	n/a
Y	In_0	C	H
	In_1	G	D
	In_2	Y_1	X_3
	In_3	Z_3	Z_3
	In_4	x	$x + y/y$
	s	0	1
	Out_0	H	n/a
	Out_1	n/a	Y_3
	Out_2	D	n/a
	$\mathrm{Out}_{\mathrm{FR}}$	n/a	Y_3^d

Point addition is computed, if y is selected for In_2 of the X processor and $x + y$ for In_4 of the Y processor during the second cycle.
Point subtraction is computed, if the selections are vice versa.

4.1. Field multipliers

Multiplication in $\mathbb{F}_{2^m}$ is the critical operation of point multiplication and much research has been published on its efficient implementation. Multiplier architectures can be categorized in three classes: bit-serial, bit-parallel, and digit-serial. A bit-serial multiplier computes one result bit in one cycle and, hence, requires m cycles. A bit-parallel multiplier computes all result bits in one cycle, but as m is large, it requires very large area. A digit-serial multiplier is a trade-off between these two extremes where D result bits are computed in one cycle resulting in a multiplication delay of $\lceil m/D \rceil$ cycles. Our point multiplication architecture described above can use any field multiplier architecture, but we chose to implement polynomial and normal basis multipliers as follows.

Polynomial basis multipliers are hardware modifications of the multiplier described in [36]. Instead of using precomputed look-up tables as in [36], our multiplier computes everything on-the-fly similarly as in [16,37]. The multiplier computes $c(x) = a(x) + b(x) \bmod p(x)$ so that, in each iteration, it computes

$$\begin{aligned} t_j(x) &= v_1(x) + v_2(x) + v_3(x), \quad \text{where} \\ v_1(x) &= x^D \sum_{i=0}^{m-D-1} t_{j+1,i} x^i \\ v_2(x) &= x^D \sum_{i=m-D}^{m-1} t_{j+1,i} x^i \bmod p(x), \quad \text{and} \\ v_3(x) &= a(x) \left(\sum_{i=0}^{D-1} b_{jD+i} x^i \right) \bmod p(x) \end{aligned} \tag{18}$$

where j is the index of the iteration (from $\lceil m/D \rceil - 1$ to 0), $b_i = 0$ for $i \geqslant m$, and $t_{\lceil m/D \rceil} = 0$. After $\lceil m/D \rceil$ iterations, $t_0(x)$ contains $c(x)$.

The multipliers are straightforward implementations of Eq. (18) which are optimized for the specific irreducible polynomials given in [25], i.e., $p(x)$ is hardwired into the design. The latency of the multiplier is given by

$$M_{\mathrm{p}} = \left\lceil \frac{m}{D} \right\rceil + 1. \tag{19}$$

Normal basis multipliers are implemented as Massey-Omura multipliers [38]. One bit c_i of the result $C = A\,B$ where $A, B, C \in \mathbb{F}_{2^m}$ is computed from A and B by using a $2m$-to-1 bit logic function, called the F-function. The F-function was constructed by using the formulae available in the appendices of [25]. It is field specific, and the same F-function is used for all output bits c_i as follows:

$$C = \sum_{i=0}^{m-1} F(A_{\lll i}, B_{\lll i}) \beta^{2^i}, \tag{20}$$

where $\lll i$ denotes cyclical left shift by i bits. A digit-serial Massey-Omura multiplier contains D parallel F-function blocks. The maximum clock frequency can be increased by pipelining the F-function blocks. As one clock cycle is required in loading the operands into the shift registers and each pipeline stage increases latency by one clock cycle, the latency becomes

$$M_n = \left\lceil \frac{m}{D} \right\rceil + c + 1 \tag{21}$$

where c is the number of pipeline stages inside the F-function blocks, i.e., $c \geqslant 0$. We use $c = 1$.

It follows directly from Eqs. (19) and (21) that, for $m = 163,233,283$, D should be chosen from the following sets of integers:

$$\mathcal{F}_{163} : \{1-15, 17, 19, 21, 24, 28, 33, 41, 55, 82, 163\}, \tag{22}$$

$$\mathcal{F}_{233} : \{1-18, 20, 22, 24, 26, 30, 34, 39, 47, 59, 78, 117, 233\}, \text{ and} \tag{23}$$

$$\mathcal{F}_{283} : \{1-19, 21, 22, 24, 26, 29, 32, 36, 41, 48, 57, 71, 95, 142, 283\} \tag{24}$$

Other values only increase area without decreasing latency.

4.2. Conversion to affine coordinates

The result point Q is mapped from $\mathcal{LD}$ to $\mathcal{A}$ at the end as shown on the last line of Algorithm 1. This mapping requires $I + 2M + S$. An inverter based on exponentiation (Fermat's Little Theorem) was chosen because it allows inversions to be computed with the same circuitry as other operations in $\mathcal{LD} \mapsto \mathcal{A}$ conversion; namely, with squarings and multiplications.

As our architecture is highly optimized for mixed coordinate point addition, its use in coordinate mapping is troublesome. Hence, a dedicated block is designed for this purpose. The block consists of a multiplier, squarer, and registers as presented in Fig. 4. The multiplier is implemented as discussed in Section 4.1 and the squarer supports successive squarings. The register is used for storing temporary results. Because an Itoh–Tsujii inversion requires $H(m - 1) - 1$ temporary variables [32] and two registers are used for storing the result point (x, y), the register consists of $H(m - 1) + 1$ m-bit registers.

We present a coordinate conversion algorithm for $m = 163$ in Algorithm 2 as an example of how coordinate conversion is computed with the above architecture. T_i and x, y are registers and t indicates that the output of the multiplier is connected directly to the inputs of the squarer and the multiplier without storing the result to the registers.

Algorithm 2. Coordinate conversion when $m = 163$

```
**Input**: $X, Y, Z \in \mathbb{F}_{2^{163}}$
**Output**: $x, y \in \mathbb{F}_{2^{163}}$ such that $(x, y) = (X/Z, Y/Z^2)$
  $T_1 \leftarrow Z^2 \times Z$
  $t \leftarrow T_1^{2^2} \times T_1$
  $t \leftarrow t^{2^4} \times t$
  $t \leftarrow t^{2^8} \times t$
  $T_2 \leftarrow t^{2^{16}} \times t$
```

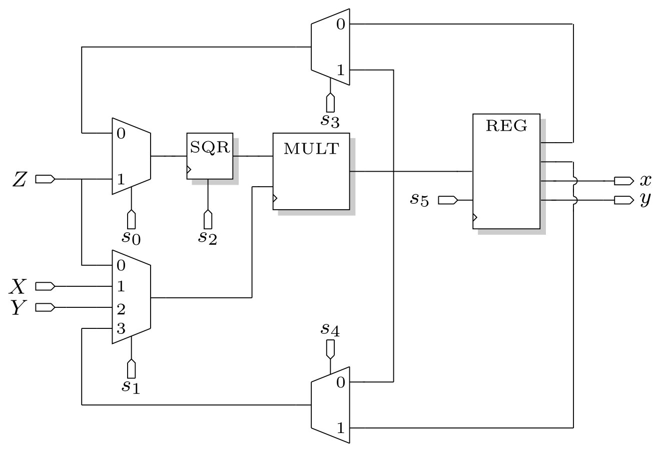

Fig. 4. Block diagram of the $\mathcal{LD}\mapsto\mathcal{A}$ converter.

$$t \leftarrow T_2^{2^{32}} \times T_2$$
$$t \leftarrow t^{2^{64}} \times t$$
$$t \leftarrow t^{2^{32}} \times T_2$$
$$T_1 \leftarrow t^{2^2} \times T_1$$
$$x \leftarrow T_1^2 \times X$$
$$y \leftarrow T_1^{2^2} \times Y$$

Latency of coordinate conversion depends on the basis. In normal basis all squarings can be performed simultaneously and the latency becomes $(\lfloor \log_2(m-1) \rfloor + H(m-1)+1)(M_n+1)$ clock cycles. In polynomial basis each squaring requires one clock cycle and the latency is $(\lfloor \log_2(m-1) \rfloor + H(m-1)+1)(M_p+1)+m+1$. The effective latency reduces by $2(M+1)$ in both cases because inversion of Z can be started as soon as the last $\mathcal{Z}_1$ is ready. Hence, coordinate conversion can be interleaved with $\mathcal{X}_1$ and $\mathcal{Y}_0$–$\mathcal{Y}_3$. The effective latencies of coordinate conversions are as follows:

$$I_p = (\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)(M_p+1) + m + 1, \quad \text{and} \tag{25}$$
$$I_n = (\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)(M_n+1) \tag{26}$$

for polynomial and normal basis, respectively.

4.3. Latency

The latency of point multiplication over polynomial basis is given by the following formula:

$$L_p = H(k)(2(M_p+1)) + I_p + \ell + 6 \tag{27}$$

where M_p and I_p are given by Eqs. (19) and (25), respectively, and ℓ is the length of k. On average, $H(k) = m/3$ and $\ell = m$. The constant delay is caused by interfacing and initialization.

Because successive squarings are performed with simple cyclical shifts in normal basis, all Frobenius maps following point addition can be performed in one clock cycle. Hence, the latency of point multiplication in normal basis is given by

$$L_n = H(k)(2(M_n+1)) + I_n + 10 \tag{28}$$

where M_n and I_n are given by Eqs. (21) and (26), respectively.

4.4. Discussion

Our architecture has several advantages over traditional architectures[2]. Table 3 shows a comparison of the traditional architecture and our architecture in terms of speed, area, and flexibility. Flexibility means that fields, curves, etc., can be changed without reconfiguring the FPGA.

As Table 3 shows, the largest advantage of our architecture is the very fast performance. The disadvantages are increased area requirements and reduced flexibility. Hence, our architecture is superior compared to the traditional architectures in high-speed applications using Koblitz curves, but it cannot be used in applications requiring low resource utilization and/or high flexibility. However, the method presented in Section 3 is more general because it can be implemented also with traditional architectures including multiple multipliers.

Circuitry for τ-adic conversion was not included in the implementations. Converters such as the ones presented in [39] should be used in practical applications. However, randomly generated τ-adic integers can be used in certain cases which removes the need for a conversion circuitry [40].

The number of operations in second (ops) can be more important than the computation time of a single point multiplication in some applications; see [15], for example. The number of ops can be increased by pipelining the architecture so that τNAF conversion, point multiplication, and coordinate conversion are separated into different pipeline stages. This approximately triples ops because all stages process a different point multiplication. In this case, smaller and slower coordinate conversion circuitry can be used.

Because our implementations target for high-speed applications, side-channel attacks are not as serious threat as in many low-cost environments such as smart cards. Currently, the implementations are vulnerable to timing attacks, i.e., they leak $H(k)$, but dummy operations could be easily introduced as a countermeasure. The use of interleaved point operations is unlikely to introduce any new side-channel threats compared to successive point operations. Anyhow side-channel resistivity should be concerned before using the implementations in any practical application where side-channel attacks are viable.

5. Results

The architecture described in Section 4 was written in VHDL. The designs were synthesized for Altera Stratix II EP2S180F1020C3 FPGA [41] by using Quartus II 6.0 SP1 design software and functionality was verified with simulations using ModelSim SE 6.1b software. We synthesized both polynomial and normal basis versions with several different multiplier sizes for K-163, and the results are given in Table 4. Because the superiority of polynomial basis is evident in Table 4, only polynomial basis versions were synthesized for K-233 and K-283. The results are given in Tables 5 and 6. The areas and maximum clock frequencies presented in the tables were given by Quartus II. Fig. 5 depicts the area and computation time of the implementations in Tables 4–6.

5.1. Discussion

The fastest implementations in Tables 4–6 compute point multiplication in 4.9 µs, 8.1 µs, and 12.1 µs on K-163, K-233, and K-283, respectively. The fastest implementations require large amounts of area, but only slightly longer point multiplication times are achievable with smaller area requirements.

Fig. 5 shows that the achieved computation time flattens when the area grows. Large amounts of additional area are needed in order to reduce the computation time even slightly and even slower performance can be achieved with a larger area. This is caused by the difficulty of place and route. When the complexities of the designs grow, the place and route becomes more difficult which degrades the results (maximum clock frequencies). Thus,

[2] By traditional architectures, we mean processor architectures including adder, squarer, multiplier(s), and storage for temporary variables, and which are controlled by microcode and/or finite state machine. See for example [9,13,14,16].

Table 3
Comparison of our architecture and traditional architectures in terms of speed, area, and flexibility

Characteristic	Traditional architecture	Our architecture
Speed	*Slow to fast.* Designer can implement both slow (and small) and fast designs by trading compactness and flexibility for speed. However, sequentiality and memory write and read delays easily become bottlenecks.	*Very fast.* The architecture is inherently very fast because the new method shortens the critical path. Computation time shortens also because additions and squarings are computed in parallel with multiplications.
Area	*Small to large.* Designer has large influence on area requirements. Control logic usually consumes more area than in our architecture.	*Medium to large.* Uses four multipliers which makes it unsuitable for low-cost applications. Multiple adders and squarers cause a small area overhead. Very simple and small control logic.
Flexibility	*Low to high.* The designer can affect flexibility on all levels. Flexible designs are slower and larger than fixed desings. The same architecture can be easily used for general and Koblitz curves.	*Low.* Field flexibility can be achieved by using flexible field multipliers and squarers but the point operation algorithm is fixed. The architecture is for Koblitz curves only.

Table 4
Results for K-163 with τNAF ($H(k) = 163/3$) on Stratix II S180C3

D	Basis	ALUTs	Regs.	ALMs	Latency	Clock (MHz)	Time (μs)
14	PB	14,960	7876	8696	1980	238.95	8.29
15	PB	15,374	7899	9192	1863	236.63	7.87
17	PB	16,265	7908	9964	1745	232.23	7.51
19	PB	17,177	7927	10,711	1627	230.47	7.06
21	PB	18,063	7908	11,550	1510	229.83	6.57
24	PB	19,596	7879	12,320	1392	215.24	6.47
28	PB	21,789	7928	14,758	1274	218.53	5.83
33	PB	23,971	7916	16,508	1157	217.53	5.32
41	PB	27,686	7937	20,525	1039	203.87	5.10
55	PB	34,604	7986	26,148	921	187.48	**4.91**
82	PB	47,302	7967	36,852	804	156.20	5.15
9	NB	19,258	14,024	13,708	2599	186.15	13.96
10	NB	20,602	14,855	15,079	2363	178.38	13.25
11	NB	21,871	15,644	16,454	2128	184.37	11.54
12	NB	23,022	16,464	17,238	2010	183.28	10.97
13	NB	24,230	17,329	18,259	1893	176.77	10.71
14	NB	25,325	18,139	19,174	1775	181.69	9.77
15	NB	24,587	18.937	21,457	1657	169.06	9.80
17	NB	26,647	20,575	23,580	1540	162.42	9.48
19	NB	28,666	22,218	25,366	1422	164.61	8.64
21	NB	30,755	23,834	27,139	1304	163.27	7.99
24	NB	33,680	26,294	29,922	1187	149.34	7.95

Table 5
Results for K-233 with τNAF ($H(k) = 233/3$) on Stratix II S180C3

D	Basis	ALUTs	Regs.	ALMs	Latency	Clock (MHz)	Time (μs)
13	PB	19,923	11,173	12,551	3780	221.78	17.04
14	PB	20,625	11,183	12,749	3614	220.22	16.41
15	PB	21,174	11,201	13,366	3449	215.70	15.99
16	PB	21,832	11,202	13,817	3284	208.33	15.76
17	PB	22,469	11,203	14,496	3118	222.07	14.04
18	PB	23,054	11,186	14,783	2953	212.86	13.87
20	PB	24,339	11,215	15,954	2788	209.21	13.32
22	PB	25,340	11,259	17,412	2622	212.13	12.36
24	PB	27,366	11,233	18,570	2457	206.23	11.91
26	PB	28,363	11,189	19,968	2292	202.51	11.32
30	PB	31,567	11,295	22,688	2126	205.72	10.34
34	PB	33,978	11,296	25,601	1961	204.04	9.61
39	PB	37,524	11,284	28,911	1796	190.19	9.44
47	PB	42,609	11,344	32,024	1630	187.48	8.70
59	PB	50,889	11,369	38,056	1465	181.06	**8.09**

Table 6
Results for K-283 with τNAF ($H(k) = 283/3$) on Stratix II S180C3

D	Basis	ALUTs	Regs.	ALMs	Latency	Clock (MHz)	Time (μs)
13	PB	24,607	13,541	15,607	5365	202.51	26.49
14	PB	25,818	13,603	16,502	5165	204.54	25.25
15	PB	26,490	13,607	17,248	4766	197.75	24.10
16	PB	27,297	13,580	17,484	4566	204.16	22.37
17	PB	28,053	13,594	18,544	4367	203.75	21.43
18	PB	28,741	13,560	18,815	4167	196.39	21.22
19	PB	29,453	13,550	19,665	3967	189.54	20.93
21	PB	31,013	13,642	21,278	3768	201.65	18.68
22	PB	31,723	13,541	21,848	3568	191.75	18.61
24	PB	33,949	13,656	23,809	3368	196.77	17.12
26	PB	35,284	13,614	25,151	3169	184.60	17.17
29	PB	38,005	13,604	27,484	2969	184.37	16.10
32	PB	40,453	13,615	29,746	2769	186.19	14.87
36	PB	43,416	13,644	33,039	2570	178.67	14.38
41	PB	47,506	13,581	36,838	2370	178.16	13.30
48	PB	53,344	13,752	39,862	2170	179.63	**12.08**

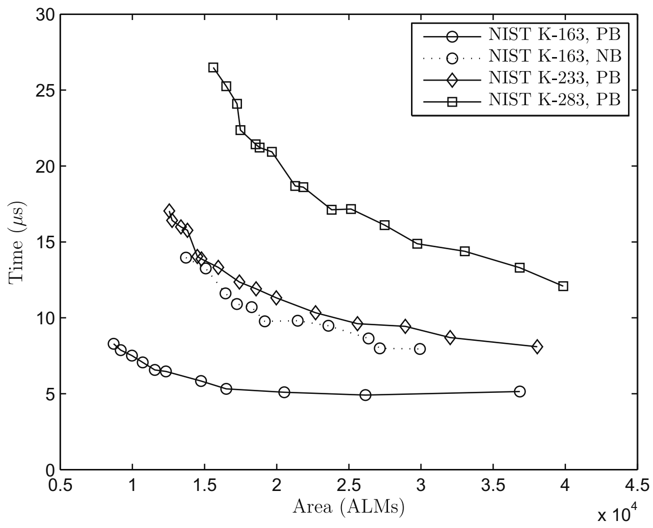

Fig. 5. Plot of areas and computation times of the implementations presented in Tables 4–6. Four multipliers are used in all implementations.

computation times can increase even though latencies decrease. The notable variation in clock frequencies (see $D = 16 - 18$ in Table 5, for example) is also caused by the unpredictable behavior of the place and route.

Table 4 shows that polynomial basis clearly outperforms normal basis. Although normal basis are clearly inferior in our architecture, this result cannot be generalized for other architectures, because our architecture has certain features which favor polynomial basis. First, the architecture includes four multipliers which quadruples the cost of the more complex multiplication in normal basis. Second, Frobenius maps are computed in parallel for the three coordinates which lessens the benefit of fast squaring in normal basis. The critical path of computing all Frobenius maps in polynomial basis is approximately m clock cycles. If only one squarer was available, Frobenius maps would require approximately $3m$ clock cycles. Hence, it is impossible to generalize our bases comparison results to an architecture which includes only one multiplier and squarer. More research on the subject is needed before drawing any general conclusions.

Table 7
Comparison of published FPGA-based implementations using Koblitz curve, K-163

Ref.	Device	Area	Time (μs)	Notes
[9]	Virtex-II	6494 slices, 6 B-RAM	35.75	NB; double-base τ-adic expansion
[14]	Stratix II	23,346 ALMs, several M512 and M4K memory blocks	28.95	NB; the fastest traditional architecture; includes three multipliers; includes a τNAF converter (approx. 930 ALMs)
[14]	Stratix II	13,472 ALMs, several M512 and M4K memory blocks	20.28	NB; four parallel processors compute a single point multiplication; includes a τNAF converter (approx. 930 ALMs)
[15]	Stratix II	67,467 ALMs, 240 M512, 305 M4K	114.2	NB; 3-term point multiplication; achieves 166,000 ops with parallel processors
[16]	Virtex-E	10,017 LUTs, 1930 FFs	75	PB
[17]	Flex 10K	n/a	45600	NB; the first Koblitz curve implementation
Our	Stratix II	23,580 ALMs	9.48	NB; comparable with [14]
Our	Stratix II	26,148 ALMs	4.91	PB

5.2. Comparisons

This comparison discusses FPGA-based implementations of point multiplication on Koblitz curves only, because our implementations are specifically designed for Koblitz curves. Hence, comparison to general curves would not be fair. FPGA implementations on Koblitz curves have been presented in [9,14–17]. Table 7 summarizes their best results. Only K-163 has been considered in the papers and, thus, we restrict this comparison to K-163. The first implementation was presented by Okada et al. [17] in 2000. Lutz and Hasan [16] presented an implementation in a Xilinx Virtex-E FPGA. Dimitrov et al. [9] presented a double-base τ-adic expansion for k and they provided experimental results in a Virtex-II FPGA. Our recent work on Koblitz curves includes [14,15]. An analysis on parallelization of point multiplication on both general and Koblitz curves was given in [14] and it was demonstrated that point multiplication can be computed efficiently with a method that splits computation for parallel processors. In [15] we studied the computation of 3-term point multiplications, i.e., $Q = k_1P_1 + k_2P_2 + k_3P_3$. The target was in maximizing the number of operations in second with parallel processing.

Fair comparison is troublesome because of the large variety of FPGAs. Comparing two designs implemented on different FPGAs is difficult because both area requirements and maximum clock frequencies depend on the FPGA. However, fair comparison to [14,15] can be performed because they use the same device. Some estimates of how Stratix II and Virtex-II compare can be given based on other contributions. A very recent work continuing the work of [9] implements the design of [9] in Stratix II where point multiplication requires 35.04 μs [42]. Hence, the speed difference of Stratix II and Virtex-II is only about 2%. Furthermore, Virtex-E has been shown to be about 10% slower than Virtex-II in implementations of elliptic curve cryptography [13]. Based on the above reasonings and Table 7 it is clear that our implementation achieving 4.91 μs is faster than any previously reported implementation. The superiority of our architecture compared to a traditional architecture (from [14]) is evident because it achieves three times faster point multiplication with a comparable amount of resources even in normal basis.

6. Conclusions and future work

We discussed point multiplication on Koblitz curves with parallel field multipliers. A novel method based on interleaved point additions was presented. It was shown to provide twice as fast point multiplication as previously presented parallel multiplier methods for Koblitz curves. The method achieves full multiplier utilization with two or four multipliers. Because also Montgomery point multiplication can be implemented efficiently with two or four multipliers, support for both general and Koblitz curves can be efficiently implemented with the same hardware. This was not possible with the existing methods.

Our new method shows that point multiplication can be computed on Koblitz curves approximately three times as fast as on general curves. With the existing methods, the benefit of using Koblitz curves decreases if several multipliers are available. However, we showed that the more multipliers are available the faster Koblitz curves are compared to general curves. Hence, our method increases the attractiveness of using Koblitz curves in high-speed applications.

We provided an extensive number of implementations supporting the practical feasibility of our method. The implementations are based on a highly optimized architecture built around four multipliers, each of which is specialized for computing certain operations. Standardized curves recommended by NIST [25] were used in the implementations; namely, the curves K-163, K-233, and K-283. We studied the suitability of polynomial basis and normal basis and concluded that polynomial basis is clearly superior to normal basis. Point multiplication times of 4.9 μs, 8.1 μs, and 12.1 μs on K-163, K-233, and K-283, respectively, were demonstrated in a Stratix II S180C3 FPGA.

The very fast point multiplication times set new challenges for designing τ-adic converters. Previously conversion times have been much shorter than any reported point multiplication times, but now conversions require approximately as long as point multiplication; see [39,40]. Hence, there is an obvious need for faster τ-adic converters.

As discussed in Section 4.4, point operation interleaving is not possible in Montgomery point multiplication. However, similar approaches may be possible in point operations on elliptic curves over $\mathbb{F}_p$. If an efficient method is found, considerable performance increases could occur and, hence, point operation interleaving requires more research.

References

[1] N. Koblitz, Elliptic curve cryptosystems, Math. Comput 48 (1987) 203–209.
[2] V. Miller, Use of elliptic curves in cryptography, in: Advances in Cryptology, CRYPTO'85, Lecture Notes in Computer Science, vol. 218, Springer, 1985, pp. 417–426.
[3] I. Blake, G. Seroussi, N. Smart, Elliptic Curves in Cryptography, London Mathematical Society Lecture Notes Series, vol. 265, Cambridge University Press, 1999.
[4] H. Eberle, N. Gura, S. Chang Shantz, V. Gupta, L. Rarick, S. Sundaram, A public-key cryptographic processor for RSA and ECC, in: Proceedings of the IEEE International Conference Application-Specific Systems, Architectures and Processors, ASAP'04, Galveston, TX, USA, 2004, pp. 98–110.
[5] J. Goodman, A. Chandrakasan, An energy-efficient reconfigurable public-key cryptography processor, IEEE J. Solid-State Circuits 36 (11) (2001) 1808–1820.
[6] F. Sozzani, G. Bertoni, S. Turcato, L. Breveglieri, A parallelized design for an elliptic curve cryptosystem coprocessor, in: Proceedings of the International Conference Information Technology: Coding and Computing, ITCC 2005, vol. 1, Las Vegas, NV, USA, 2005, pp. 626–630.
[7] T. Wollinger, J. Guajardo, C. Paar, Security on FPGAs: state-of-the-art implementations and attacks, ACM Trans. Embed. Comput. Syst. 3 (3) (2004) 534–574.
[8] B. Ansari, M.A. Hasan, High performance architecture of elliptic curve scalar multiplication, Tech. Rep. CORR 2006-01, University of Waterloo, Canada, 2006.

[9] V.S. Dimitrov, K.U. Järvinen, M.J. Jacobson, W.F. Chan, Z. Huang, FPGA implementation of point multiplication on Koblitz curves using Kleinian integers, in: Cryptographic Hardware and Embedded Systems, CHES 2006, Lecture Notes in Computer Science, vol. 4249, Springer, 2006, pp. 445–459.
[10] H. Eberle, N. Gura, S. Chang-Shantz, A cryptographic processor for arbitrary elliptic curves over GF(2^m), in: Proceedings of the IEEE International Conference Application-Specific Systems, Architectures, and Processors, ASAP'03, The Hague, The Netherlands, 2003, pp. 444–454.
[11] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, J. Teich, Reconfigurable implementation of elliptic curve crypto algorithms, in: Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS 2002, Reconfigurable Architectures Workshop, RAW 2002, Ft. Lauderdale, FL, USA, 2002, pp. 157–164.
[12] M. Bednara, M. Daldrup, J. Teich, J. von zur Gathen, J. Shokrollahi, Tradeoff analysis of FPGA based elliptic curve cryptography, in: Proceedings of the IEEE International Symposium Circuits and Systems, ISCAS 2002, vol. 5, Phoenix-Scottdale, AZ, USA, 2002, pp. 797–800.
[13] R.C.C. Cheung, N.J. Telle, W. Luk, P.Y.K. Cheung, Customizable elliptic curve cryptosystem, IEEE Trans. VLSI Syst. 13 (2005) 1048–1059.
[14] K. Järvinen, J. Skyttä, On parallelization of high-speed processors for elliptic curve cryptography, IEEE Trans. VLSI Syst. 16 (2008) 1162–1175.
[15] K. Järvinen, J. Forsten, J. Skyttä, FPGA design of self-certified signature verification on Koblitz curves, in: Cryptographic Hardware and Embedded Systems, CHES 2007, Lecture Notes in Computer Science, vol. 4727, Springer, 2007, pp. 256–271.
[16] J. Lutz, A. Hasan, High performance FPGA based elliptic curve cryptographic co-processor, in: Proceedings of the International Conference Information Technology: Coding and Computing, ITCC 2004, vol. 2, Las Vegas, NV, USA, 2004, pp. 486–492.
[17] S. Okada, N. Torii, K. Itoh, M. Takenaka, Implementation of elliptic curve cryptographic coprocessor over GF(2^m) on an FPGA, in: Cryptographic Hardware and Embedded Systems, CHES 2000, Lecture Notes in Computer Science, vol. 1965, Springer, 2000, pp. 25–40.
[18] F. Rodríguez-Henríquez, N.A. Saqib, A. Díaz-Pérez, A fast parallel implementation of elliptic curve point multiplication over GF(2^m), Microprocess. Microsy 28 (5–6) (2004) 329–339.
[19] K. Sakiyama, L. Batina, B. Preneel, I. Verbauwhede, Superscalar coprocessor for high-speed curve-based cryptography, in: Cryptographic Hardware and Embedded Systems, CHES 2006, Lecture Notes in Computer Science, vol. 4249, Springer, 2006, pp. 415–429.
[20] C. Shu, K. Gaj, T. El-Ghazawi, Low latency elliptic curve cryptography accelerators for NIST curves over binary fields, in: Proceedings of the IEEE International Conference Field Programmable Technology, FPT 2005, Singapore, 2005, pp. 309–310.
[21] G. Meurice de Dormale, J.-J. Quisquater, High-speed hardware implementations of elliptic curve cryptography: a survey, J. Syst. Architect. 53 (2–3) (2007) 72–84.
[22] D. Hankerson, A. Menezes, S. Vanstone, Guide to Elliptic Curve Cryptography, Springer, 2004.
[23] J. López, R. Dahab, Fast multiplication on elliptic curves over GF(2^m) without precomputation, in: Cryptographic Hardware and Embedded Systems, CHES 1999, Lecture Notes in Computer Science, vol. 1717, Springer, 1999, pp. 316–317.
[24] N. Koblitz, CM-curves with good cryptographic properties, in: Advances in Cryptology, CRYPTO'91, Lecture Notes in Computer Science, vol. 576, Springer, 1991, pp. 279–287.
[25] National Institute of Standards and Technology (NIST), Digital signature standard (DSS), Federal Information Processing Standard, FIPS PUB 186-2, January 27, 2000.
[26] Certicom Research, SEC 1: Elliptic curve cryptography, Standards for Efficient Cryptography, September 20, 2000.
[27] Certicom Research, SEC 2: Recommended elliptic curve domain parameters, Standards for Efficient Cryptography, September 20, 2000.
[28] P.L. Montgomery, Speeding the Pollard and elliptic curve methods of factorization, Math. Comput. 48 (177) (1987) 243–264.
[29] C. Doche, T. Lange, Arithmetic of elliptic curves, in: H. Cohen, G. Frey (Eds.), Handbook of Elliptic and Hyperelliptic Curve Cryptography, Chapman & Hall/CRC, 2006, pp. 267–302. Ch. 13.
[30] J. López, R. Dahab, Improved algorithms for elliptic curve arithmetic in GF(2^n), in: Selected Areas in Cryptography, SAC'98, Lecture Notes in Computer Science, vol. 1556, Springer, 1998, pp. 201–212.
[31] E. Al-Daoud, R. Mahmod, M. Rushdan, A. Kilicman, A new addition formula for elliptic curves over GF(2^n), IEEE Trans. Comput. 51 (8) (2002) 972–975.
[32] T. Itoh, S. Tsujii, A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases, Inform. Comput. 78 (3) (1988) 171–177.
[33] J.A. Solinas, Efficient arithmetic on Koblitz curves, Des. Codes Cryptogr. 19 (2–3) (2000) 195–249.
[34] T. Lange, A note on López-Dahab coordinates, Cryptology ePrint Archive, Report 2004/323, <http://eprint.iacr.org/>, 2004.
[35] G. Meurice de Dormale, P. Bulens, J.-J. Quisquater, Collision search for elliptic curve discrete logarithm over GF(2^m) with FPGA, in: Cryptographic Hardware and Embedded Systems, CHES 2007, Lecture Notes in Computer Science, vol. 4727, Springer, 2007, pp. 378–393.
[36] M.A. Hasan, Look-up table-based large finite field multiplication in memory constrained cryptosystems, IEEE Trans. Comput. 49 (7) (2000) 749–758.
[37] J. Lutz, High performance elliptic curve cryptographic co-processor, Master's Thesis, University of Waterloo, 2003.
[38] C.C. Wang, T.K. Troung, H.M. Shao, L.J. Deutsch, J.K. Omura, I.S. Reed, VLSI architectures for computing multiplications and inverses in GF(2^m), IEEE Trans. Comput. 34 (8) (1985) 709–717.
[39] K. Järvinen, J. Forsten, J. Skyttä, Efficient circuitry for computing τ-adic non-adjacent form, in: Proceedings of the IEEE International Conference Electronics, Circuits and Systems, ICECS 2006, Nice, France, 2006, pp. 232–235.
[40] B.B. Brumley, K. Järvinen, Koblitz curves and integer equivalents of Frobenius expansions, in: Selected Areas in Cryptography, SAC 2007, Lecture Notes in Computer Science, vol. 4876, Springer, 2007, pp. 126–137.
[41] Altera, Stratix II Device Handbook, vol. 1–2, ver.4.1, <http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf>, April 2006.
[42] V.S. Dimitrov, K.U. Järvinen, M.J. Jacobson, W.F. Chan, Z. Huang, Provably sublinear point multiplication on Koblitz curves and its hardware implementation, IEEE Trans. Comput., in press, doi:10.1109/TC.2008.65.

Kimmo Järvinen was born in Turku, Finland, in 1979. He received M.Sc. (Tech.) in Electrical Engineering degree from the Helsinki University of Technology (TKK) in 2003. He has been with the Signal Processing Laboratory at TKK since 2002 and in the Graduate School in Electronics, Telecommunications and Automation (GETA) since 2004. He is currently pursuing D.Sc. (Tech.) degree. His research interests include hardware realization of cryptographic algorithms, secret-key and public-key cryptographic algorithms, especially elliptic curve cryptography, and FPGAs.

Jorma Skyttä was born in Helsinki, Finland, in 1957. He received M.Sc. (Tech.), Lic. Tech., and D.Sc. (Tech.) degrees from the Helsinki University of Technology (TKK) in 1981, 1984, and 1985, respectively. He has been with the Signal Processing Laboratory at TKK since 1989 as Associate Professor and Professor since 1998. His research interest includes realization of digital computation, especially digital signal processing applications and implementation of cryptographic systems utilizing FPGAs, and FPGA-based computational architectures.