A Fast and Compact FPGA Implementation of Elliptic Curve Cryptography Using Lambda Coordinates

Burak Gövem[1], Kimmo Järvinen[1], Kris Aerts[2], Ingrid Verbauwhede[1], and Nele Mentens[1,2]

[1] KU Leuven ESAT/COSIC and iMinds
Kasteelpark Arenberg 10 bus 2452, B-3001 Leuven-Heverlee, Belgium
[2] KU Leuven Technology Campus Diepenbeek, ES&S, Belgium
Firstname.Lastname@esat.kuleuven.be

Abstract. Elliptic curve cryptography (ECC) provides high security with shorter keys than other public-key cryptosystems and it has been successfully used in security critical embedded systems. We present an FPGA-based coprocessor that communicates with the host processor via a 32-bit bus. It implements ECC over an elliptic curve that offers roughly 128-bit security. It is the first hardware implementation that uses the recently introduced lambda coordinates and the Galbraith-Lin-Scott (GLS) technique with fast endomorphisms. One scalar multiplication requires 65,000 clock cycles with a maximum clock frequency of 274 MHz on a Xilinx Virtex-5 FPGA, which gives a computation time of 0.24 ms. The area utilization is 1552 slices and 4 BlockRAMs. Our coprocessor compares favorably to other published works both in terms of speed and area, which makes it a good choice for embedded systems that require high-security public-key cryptography.

1 Introduction

Many embedded systems are used in applications where security and safety are of utmost importance. Such security-critical embedded systems include airplanes, cars, medical devices, home automation systems, military devices, etc. They require that confidentiality, integrity, and authenticity are ensured by using strong cryptography. Cryptography is often computationally demanding and efficient implementation of cryptographic computations is a topic that has been an active research field during the last couple of decades. In particular, public-key cryptosystems are challenging to implement efficiently and securely in embedded systems because they are computationally more demanding than many other forms of encryption. Public-key cryptosystems are essential parts of a variety of cryptosystems because they are required, for example, for computing digital signatures. Therefore, techniques for their fast computation with small amounts of resources are needed in embedded systems in practice.

Elliptic curve cryptography (ECC) is a type of public-key cryptography that was introduced in the mid-1980s. ECC has many benefits compared to other

forms of public-key cryptography. The main benefit is that high security levels can be achieved with shorter keys than for other public-key cryptosystems (such as RSA, Diffie-Hellman, ElGamal, etc.). ECC uses $2n$-bit keys for achieving roughly n bits of security. For example, RSA requires significantly longer keys: 1,024 and 3,072 bits for 80-bit and 128-bit security levels, respectively [34]. ECC implementations have also proven to be faster than RSA implementations [15]. ECC is also widely included in multiple standards, which is a significant advantage in commercial applications.

Field-programmable gate arrays (FPGA) have been popular implementation platforms for cryptography and a plethora of ECC implementations for FPGAs are available in the literature. While many of them target primarily to hardware acceleration of ECC by optimizing speed with very loose area constraints (see, e.g., [24, 1, 20, 11, 4]), there also exist many implementations that are suitable for security-critical embedded systems including, e.g., [41, 28, 26, 42]. In them, the primary optimization target is typically either area or speed-area ratio. A vast majority of such designs has been introduced for the 80-bit security level, which should have been used only up to 2010 as recommended by the National Institute of Standards and Technology (NIST) of the United States [33]. Most of the publications target the NIST curves specified in [35]. These curves date back to the 1990s and they cannot utilize certain state-of-the-art optimizations that have been introduced in the recent years. Several new curves have been introduced allowing more efficient computations (see, e.g., [14, 13, 7, 9, 8, 17, 37, 10]). Although many studies about software performance of these curves are available, hardware implementations are still largely missing from the literature (for some exceptions, see, e.g., [3, 2, 5, 6, 16, 40]). In particular, to the best of our knowledge, there are no hardware implementations of the new λ-coordinates [37] combined with the Galbraith-Lin-Scott (GLS) technique for binary curves [17] which was shown to be very efficient in software [37]. The implementation in [2] uses the same curve without λ-coordinates, focusing on speed maximization with very loose area constraints. It is widely known that there is often a difference between efficiency in software and hardware and some curves that are fast in software may not be as efficient in hardware, and vice versa. Good examples are prime and binary curves, of which the former are better in software and the latter in hardware. Also NIST has shown interest in the new curves and techniques and there appears to be considerations to standardize new elliptic curves (see, e.g., the call for papers of the NIST workshop on elliptic curves [36]). Hence, it is important to shed light on hardware performance of the new curves and techniques in order to complete the picture about their efficiency.

In this paper, we present an FPGA coprocessor for ECC which is designed primarily for security-critical embedded systems. Our coprocessor communicates with a host processor via a 32-bit interface which allows easy integration to various systems. The coprocessor implements ECC for the 128-bit security level which matches, e.g., the security offered by the 128-bit version of the Advanced Encryption Standard (AES) [32]. Most other publications consider the significantly less secure 80-bit security level as discussed above. The coprocessor is

designed so that it is both fast and compact and, thus, meets the requirements of various security-critical embedded systems. Our implementation uses the elliptic curve and parameters as well as many of the state-of-the-art optimizations introduced by Oliveira et al. [37] in 2013. To the best of our knowledge, our implementation is the first FPGA-based ECC implementation that uses λ-coordinates from [37] and the GLV/GLS technique from [14, 13, 17]. We compiled our architecture for Xilinx Virtex-4, Virtex-5, and Spartan-6 FPGAs and the results show that our coprocessor compares favorably with the related work available in the literature although it offers a significantly higher security level. Our results also show that λ-coordinates and the GLS technique provide good results not only in software but also in hardware.

The paper is structured as follows. Section 2 presents the preliminaries of ECC and the algorithms we implement in our coprocessor. Section 3 describes the architecture of our coprocessor. Section 4 presents the results on Xilinx FPGAs and compares them to other relevant ECC implementations available in the literature. We end with conclusions and discussion on certain topics for future research in Section 5.

2 Preliminaries

This section provides background on ECC in general and on GLS curves and λ-coordinates that we implement in this paper in particular.

2.1 Elliptic Curve Cryptography

The use of elliptic curves for public-key cryptography was independently proposed by Victor Miller [31] and Neal Koblitz [21] in the mid-1980s. ECC achieves high security levels with significantly shorter key lengths than other public-key cryptosystems such as RSA or ElGamal. Hence, ECC has become a popular choice for public-key cryptography especially in embedded systems.

Elliptic curves defined over a finite field $\mathbb{F}_q$ are used in cryptography. The points (x, y) that satisfy the equation of an elliptic curve combined with a special point called the point-at-infinity $\mathcal{O}$ form an additive Abelian group E, where $\mathcal{O}$ is the zero element. Let $P_1, P_2 \in E$. The group operation $P_1 + P_2$ is called point addition if $P_1 \neq \pm P_2$ and point doubling if $P_1 = P_2$. The most important operation of every elliptic curve cryptosystem is the scalar multiplication:

$$Q = kP = \underbrace{P + P + \ldots + P}_{k \text{ times}} \tag{1}$$

where k is an integer in the interval $[1, r-1]$ where r is the order of P (the smallest positive integer for which $rP = \mathcal{O}$). The security of ECC is based on the computational difficulty of the elliptic curve discrete logarithm problem (ECDLP), which is the problem of finding k when given Q and P. The ECDLP is believed to be infeasible to solve if the parameters of the system are chosen properly

(similarly as integer factorization in the case of RSA). A secure elliptic curve over a $2m$-bit finite field is believed to offer roughly m bits of security. Hence, ECC offers roughly 128-bit security with 256-bit keys whereas, for example, RSA would require approximately 3,072-bit keys [34].

Computing (1) consists of several hierarchical levels, which are depicted in the ECC pyramid of Fig. 1. The scalar multiplication algorithm is on the top and it computes (1) with a series of point arithmetic operations (typically, point additions and point doublings). The algorithms that implement the point operations with series of finite field operations are in the middle. The algorithms for computing finite field operations including multiplication, addition (subtraction), and inversion (division) are in the bottom. In the following, we discuss the hierarchical levels and provide descriptions of our choices for implementing them from the bottom to the top.

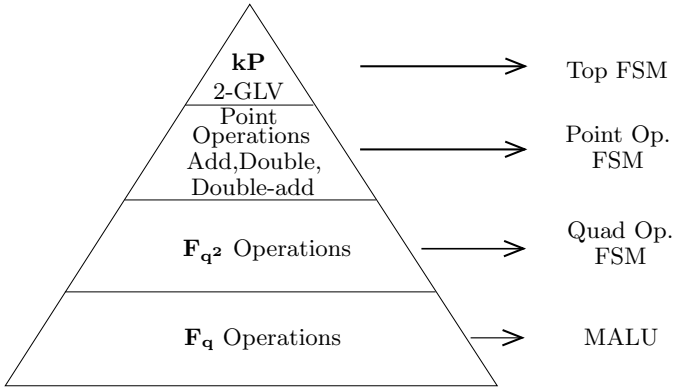

Fig. 1. ECC hierarchical pyramid. The components in our architecture (see Sect. 3) for implementing the hierarchical levels of the pyramid are shown on the right.

2.2 Finite Field Arithmetic

Either prime fields, where q is a prime p, or binary extension fields, where $q = 2^m$, are typically used for ECC. Prime fields are more commonly used in software implementations. Binary fields allow significantly more efficient implementations in hardware because they employ carry-free arithmetic operations. The inclusion of carry-free instructions in modern processors has enabled extremely fast implementations using binary fields also in software [43]. Elliptic curves defined over prime and binary fields are called prime and binary curves, respectively.

In this paper, we follow the approach of [37] and use the binary field $\mathbb{F}_{2^{254}}$ which can be constructed as a quadratic extension of the binary extension field $\mathbb{F}_{2^{127}}$ by using the irreducible polynomial $g(u) = u^2 + u + 1$. That is, we set $\mathbb{F}_{2^{127}}[u] \,/\, u^2 + u + 1$. An element $a \in \mathbb{F}_{2^{254}}$ is represented as $a = a_0 + a_1 u$ where

$a_0, a_1 \in \mathbb{F}_{2^{127}}$. Arithmetic operations in $\mathbb{F}_{2^{254}}$ can be decomposed into operations in $\mathbb{F}_{2^{127}}$ and they are computed as follows [37]:

$$a + b = a_0 + b_0 + (a_1 + b_1)u \tag{2}$$

$$a \times b = (a_0 b_0 + a_1 b_1) + (a_0 b_1 + a_1 b_0 + a_1 b_1)u \tag{3}$$

$$a^2 = a_0^2 + a_1^2 + a_1^2 u \tag{4}$$

$$a^{-1} = (a_0 + a_1)t^{-1} + a_1 t^{-1} u \tag{5}$$

where $t = a_0 a_1 + a_0^2 + a_1^2$. Let $\mathbf{a}$, $\mathbf{m}$, $\mathbf{s}$, and $\mathbf{i}$ denote the costs of addition, multiplication, squaring, and inversion in $\mathbb{F}_{2^{127}}$ and let $\mathbf{A}$, $\mathbf{M}$, $\mathbf{S}$, and $\mathbf{I}$ denote the costs of the respective operations in $\mathbb{F}_{2^{254}}$. The above equations give that $\mathbf{A} = 2\mathbf{a}$, $\mathbf{M} = 4\mathbf{m} + 3\mathbf{a}$, $\mathbf{S} = 2\mathbf{s} + \mathbf{a}$, and $\mathbf{I} = \mathbf{i} + 3\mathbf{m} + 2\mathbf{s} + 3\mathbf{a}$.

We construct $\mathbb{F}_{2^{127}}$ by setting $\mathbb{F}_2[x] \,/\, p(x)$ where $p(x)$ is the irreducible trinomial $p(x) = x^{127} + x^{63} + 1$. Addition in $\mathbb{F}_{2^{127}}$ is a bitwise exclusive-or (xor) of the bit vectors representing the elements. Multiplication is carried out by computing a multiplication of polynomials in $\mathbb{F}_2[x]$ followed by a reduction modulo $p(x)$. Because squaring in $\mathbb{F}_2[x]$ can be performed by adding zeros between each bit of the bit vector, squaring in $\mathbb{F}_{2^{127}}$ contains only rewiring followed by the reduction modulo $p(x)$ when implemented in hardware. Inversion can be computed as an exponentiation consisting of multiplications and squarings in $\mathbb{F}_{2^{127}}$ by using the Itoh-Tsujii algorithm [19]. We use a variant of the Itoh-Tsujii inversion from [37] that utilizes the optimal addition chain $(1, 2, 3, 6, 12, 24, 48, 96, 120, 126)$ and requires $126\mathbf{s} + 9\mathbf{m}$ in $\mathbb{F}_{2^{127}}$.

2.3 Point Representation with λ-Coordinates

Point addition and point doubling are the basic point operations. If the points are represented in affine coordinates by using two coordinates (x, y), then both point addition and point doubling require an inversion, which is a very expensive operation as shown in Section 2.2. Hence, projective coordinates, where points are represented with three coordinates as (X, Y, Z), are commonly used in practical implementations of ECC. They allow computing point additions and point doublings without inversions (but with an increased number of other operations). A single inversion is required in the end of computing (1) in order to obtain the affine coordinates of the result point Q. In the case of binary curves, popular choices have been standard projective coordinates, where $x = X/Z$ and $y = Y/Z$, and López-Dahab (LD) coordinates [27], where $x = X/Z$ and $y = Y/Z^2$.

In 2013, Oliveira et al. [37] proposed a new coordinate system called λ-coordinates, where points are represented as (x, λ) so that $\lambda = x + \frac{y}{x}$. We refer to this coordinate system as affine λ-coordinates. The projective version of λ-coordinates represents a point with three coordinates (X, L, Z) so that $x = X/Z$ and $\lambda = L/Z$. They result in the fastest formulae that are currently available for computing point arithmetic on binary Weierstrass curves. A point addition and point doubling require (excluding additions) $8\mathbf{M} + 2\mathbf{S}$ and $5\mathbf{M} + 4\mathbf{S}$ (including one multiplication by a constant), respectively. One of the main benefits of λ-coordinates compared to LD coordinates is that they allow efficient combination

Algorithm 1 Double-and-add (left-to-right)

Require: Scalar $k = \sum_{i=0}^{n-1} k_i 2^i \in [0, r-1]$ with $k_{n-1} = 1$, base point P
Ensure: Result point $Q = kP$
  $Q \leftarrow P$
  for $i = n-2$ **down to** 0 **do**
    $Q \leftarrow 2Q$
    if $k_i = 1$ **then**
      $Q \leftarrow Q + P$
    end if
  end for
  return Q

of point doubling and point addition operations. Computing $2P_1 + P_2$ requires only $11\mathbf{M} + 6\mathbf{S}$ (including one multiplication with a constant).

2.4 Scalar Multiplication on the GLS Curves

In this paper, we use the same curve that was used in [37]. It is a Weierstrass curve over $\mathbb{F}_{2^{254}}$ defined by the following equation:

$$y^2 + xy = x^3 + ax^2 + b \tag{6}$$

where $a = a_0 + a_1 u, b = b_0 + b_1 u \in \mathbb{F}_{2^{254}}$ such that $a_0 = b_1 = 0$, $a_1 = 1$, and b_0 is a specific element in $\mathbb{F}_{2^{127}}$.

Scalar multiplication defined by (1) can be computed with a series of point additions and point doublings. The simplest option is to use the double-and-add algorithm given in Algorithm 1. It scans through the bits of k and performs a point doubling for every bit and an additional point addition if the bit is one. Let k be an n-bit integer and let $h(k)$ be its Hamming weight (the number of ones in the binary expansion). Then, one scalar multiplication requires $n-1$ point doublings and $h(k)-1$ point additions, where $h(k) \approx n/2$.

One way of improving the speed of scalar multiplications is to utilize efficiently computable endomorphisms. Menezes and Vanstone [30] showed how point doublings can be replaced with the Frobenius endomorphisms $(x, y) \mapsto (x^2, y^2)$ on certain supersingular elliptic curves over $\mathbb{F}_{2^m}$, but these curves were found to be cryptographically weak [29]. In 1991, Koblitz [22] introduced a secure class of nonsupersingular elliptic curves over $\mathbb{F}_{2^m}$ which have the advantage of the Frobenius endomorphisms after certain conversions are computed for the scalar k. These curves are commonly known as Koblitz curves and they are nowadays included in many standards (e.g., in [35]). In 2001, Gallant, Lambert, and Vanstone (GLV) [14] introduced a specific class of elliptic curves over $\mathbb{F}_p$ which allows utilizing efficiently computable endomorphisms also for prime curves. Galbraith, Lin, and Scott (GLS) [13] generalized the GLV technique to a broader class of elliptic curves defined over $\mathbb{F}_{p^2}$. The GLS curves were generalized for binary curves over $\mathbb{F}_{2^{2m}}$ by Hankerson, Karabina, and Menezes [17]. In

this paper, we focus on these variants of the GLS curves and, in particular, the curve considered by Oliveira et al. in [37].

The GLS technique allows splitting the computation of (1) with an n-bit k into $k_1P + k_2\psi(P)$ where k_1 and k_2 are approximately $n/2$-bit integers. That is, instead of the single scalar multiplication, one computes a sum of two smaller scalar multiplications, which can be computed efficiently with the so called Shamir's trick (see below). We skip the mathematical subtleties and merely state that the efficiently computable endomorphism ψ of the GLS technique is based on a composition of the Frobenius endomorophism and endomorphisms between E and its quadratic twist. Interested readers can find details, e.g., from [37, 17, 14, 13]. We use $\psi(P)$ as defined in [37] as follows:

$$\psi : (x, \lambda) \mapsto ((x_0 + x_1) + x_1 u, (\lambda_0 + \lambda_1) + (\lambda_1 + 1)u) \tag{7}$$

where $x = x_0 + x_1 u$ and $\lambda = \lambda_0 + \lambda_1 u$ with $x_0, x_1, \lambda_0, \lambda_1 \in \mathbb{F}_{2^{127}}$. Hence, $\psi(P)$ requires only three additions ($3\mathbf{a}$) in $\mathbb{F}_{2^{127}}$.

The integer k needs to be decomposed into $n/2$-bit k_1 and k_2 such that $k \equiv k_1 + k_2\delta \pmod{r}$ where δ is an integer such that $\psi(P) = \delta P$ for all $P \in E$. Such a decomposition can be found by using techniques for finding the GLV decomposition given in [14]. The decomposition algorithm can be simplified for specific curve parameters and we use the decomposition algorithm used by Oliveira et al. [37] in the C code that is publicly available[3]. It finds k_1 and k_2 by computing:

$$k_1 = k_l + k_h - \beta\lfloor (k\beta)/2^{254} \rfloor \tag{8}$$

$$k_2 = k_h\beta - (2^{127} - 1)\lfloor (k\beta)/2^{254} \rfloor \tag{9}$$

where β is a 64-bit constant specific for the curve and k_l and k_h are the lowest and highest 127-bit words of the 254-bit k.

Shamir's trick (see, e.g., [18]) is a technique that allows evaluating a sum of two scalar multiplications $k_1P + k_2\psi(P)$ simultaneously. We call this operation double scalar multiplication. Let k_1 and k_2 be $n/2$-bit integers. If the double scalar multiplication is computed with two separate scalar multiplications, then it requires $n-2$ point doublings and $h(k_1)+h(k_2)-1 \approx n/2-1$ point additions. Shamir's trick arranges k_1 and k_2 into a $2 \times n/2$-bit matrix and precomputes the point $P + \psi(P)$. The double scalar multiplication is computed by scanning through the columns of the matrix. A point doubling is computed for every column and a point addition is computed for all nonzero columns. If the column is $\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$ or $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$, then one adds either P, $\psi(P)$ or $P+\psi(P)$, respectively. Hence, Shamir's trick requires $n/2 - 1$ point doublings and, on average, $\frac{3}{8}n$ point additions for the double scalar multiplication.

In [37], Oliveira et al. used a parallelization technique that splits the scalar multiplication in two parallel computations: one based on the double-and-add and the other on halve-and-add. Halve-and-add computes point halvings $Q \leftarrow \frac{1}{2}Q$ instead of point doublings. They also utilized the window nonadjacent form

[3] https://github.com/floodyberry/supercop/tree/master/crypto_dh/gls254

(w-NAF) for representing k_1 and k_2 in the GLV encoding which leads to a smaller number of point additions but requires precomputations and extra storage for the precomputed points. We decided not to use these optimizations in our implementation because implementing them would lead to a significant growth of the control logic and the parallelization technique would also require another unit for field arithmetic (see MALU in Section 3.4).

To summarize, we implement the scalar multiplication by using the GLS technique which splits an n-bit k into $n/2$-bit k_1 and k_2 so that both are given in standard binary representation. We precompute $P+\psi(P)$. We then use Shamir's trick for evaluating the double scalar multiplication by computing point doublings and combined point doublings and additions. The point arithmetic is performed in projective λ-coordinates by using the formulae from [37]. Finite field arithmetic is computed in $\mathbb{F}_{2^{254}}$ by decomposing the operations into operations in $\mathbb{F}_{2^{127}}$ as shown in (2)–(5).

3 Architecture

We present a coprocessor architecture for FPGAs that implements (1) using λ-coordinates and the fast GLS endomorphism. In this architecture, the finite field processing unit (FFPU) performs operations in $\mathbb{F}_{2^{127}}$ and three control units (i.e., finite state machines (FSM)) drive the FFPU in a hierarchical manner according to the ECC hierarchy shown in Figure 1. This provides a natural way to decompose the complex control logic required for computing (1) into a set of smaller FSM, which can be implemented efficiently in FPGAs. Additionally, smaller processing and control units for integer arithmetic perform the scalar decomposition. A register file and block RAMs (BRAM) are used for storing temporary results. The register file stores frequently used temporary variables. The base point, the result of a scalar multiplication and the curve constant a are also stored in the BRAM. In the following, we describe the architecture of the coprocessor in a hierachical manner according to Figure 1.

3.1 Top Level FSM

Our implementation is designed to be used as an ECC coprocessor. The architecture of the coprocessor is shown in Figure 2. The host processor initiates the coprocessor by sending the base point P and scalar k over a 32-bit bus. The base point is stored in the BRAM and the scalar is stored in the registers. Next, the coprocessor starts the precomputation. The scalar multiplication FSM (the Top FSM in Figure 2) manages the point operation FSM, shifts the scalar registers k_1 and k_2, and determines the next point operation. The scalar multiplication FSM also organizes precomputations, which are the scalar decomposition, computation of the endomorphism $\psi(P)$, the point addition for computing $P+\psi(P)$, finding the first nonzero column of the scalar matrix, and performing the coordinate transformations. First, $\psi(P)$ is calculated and stored in the BRAM and, then, it is added to P. Since the combined point doubling

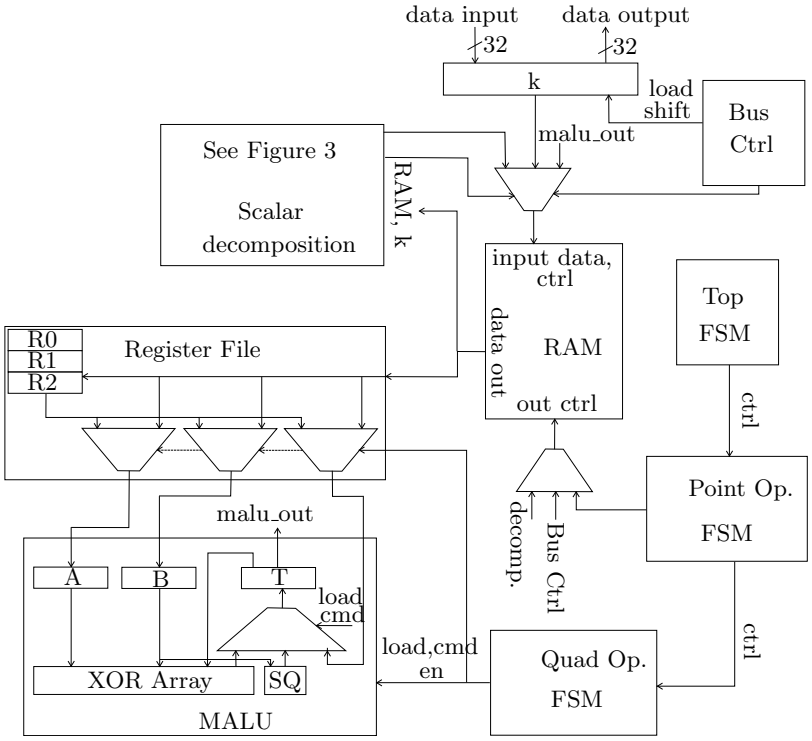

Fig. 2. Architecture

and point addition requires the other input point to be given in affine λ coordinates, the point $P + \psi(P)$ is converted to affine λ-coordinates and stored in the BRAM. After that, the scalar decomposition that computes (8) and (9) starts and when it is ready, the precomputation ends with finding the first nonzero column of the scalar matrix. Because all points and the scalars are ready, the scalar multiplication starts after this. As discussed in Section 2.4, the double scalar multiplication $k_1P + k_2\psi(P)$ is implemented using Shamir's trick. The accumulator point Q is initialized with P if the first nonzero column is $\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$, with $\psi(P)$ if $\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$, and with $P + \psi(P)$ if $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$. After this, either point doublings (for zero columns) or the combined point doublings and additions (for nonzero columns) are performed depending on the bits of k_1 and k_2 and the points to be added are determined as above. After the scalar multiplication is finished, the result point Q is first converted to affine λ-coordinates and then to affine coordinates and it is stored into a specific location in the BRAM, where it is available to the host processor. The scalar multiplication algorithm implemented by the coprocessor is shown in Algorithm 2.

Algorithm 2 Scalar multiplication on a GLS curve, $Q = kP$

Require: Scalars $k_1 = (k_{1,n_1-1} \dots k_{1,0})$, $k_2 = (k_{2,n_2-1} \dots k_{2,0})$, base point P
Ensure: Result point $Q = kP = k_1P + k_2\psi(P)$

```
P1 ← ψ(P)                                              // 3a in F_{2^127}
P2 ← P + P1                                            // Algorithm 3
n ← max(n1, n2)
if k1,n−1 = 1 and k2,n−1 = 1 then
   Q ← P2
else if k1,n−1 = 0 and k2,n−1 = 1 then
   Q ← P1
else if k1,n−1 = 1 and k2,n−1 = 0 then
   Q ← P
end if
for i = n − 2 down to 0 do
   if k1,i = 1 and k2,i = 1 then
      Q ← 2Q + P2                                      // Algorithm 5
   else if k1,i = 0 and k2,i = 1 then
      Q ← 2Q + P1                                      // Algorithm 5
   else if k1,i = 1 and k2,i = 0 then
      Q ← 2Q + P                                       // Algorithm 5
   else
      Q ← 2Q                                           // Algorithm 4
   end if
end for
Q = (x, y) ← affine(Q)                                 // I + 3M + A in F_{2^254}
return Q
```

3.2 Point Operations FSM

Point operations (point doubling, point addition and combined point doubling and addition) and coordinate conversion are implemented in the point operation FSM. Point addition, point doubling and combined point doubling and addition require $5\mathbf{M} + 2\mathbf{S} + 5\mathbf{A}$, $5\mathbf{M} + 4\mathbf{S} + 5\mathbf{A}$, and $11\mathbf{M} + 6\mathbf{S} + 9\mathbf{A}$ operations in $\mathbb{F}_{2^{254}}$, respectively. Affine coordinates to affine λ-coordinates, affine λ-coordinates to affine coordinates, and projective λ-coordinates to affine λ-coordinates conversions require $\mathbf{I} + \mathbf{M} + \mathbf{A}$, $\mathbf{M} + \mathbf{A}$ and $\mathbf{I} + 2\mathbf{M}$, respectively. One of the multiplications in both point doubling and combined point doubling and addition is a multiplication by a constant (the curve parameter a).

The point operations FSM fetches input operands for an operation in $\mathbb{F}_{2^{254}}$ and writes them into registers of the register file. After the operation in $\mathbb{F}_{2^{254}}$ is finished, the point operation FSM writes the result of the operation to the BRAM unless the result is required only for the next operation. In that case, the writing is skipped and the point operation FSM proceeds to the next operation that will operate directly on the result in the register file. Results are written to the register file by default. For some cases results are both needed in the next operation and in later operations. Therefore, the result is stored in the BRAM to be used in the later operations and the next operation uses the result in the

register file in order to save clock cycles. Temporary variable $R3$ that is shown in Algorithm 3, 4 and 5 is a register in the register file and all other temporary variables are in the BRAM.

Point addition is used only once when $P + \psi(P)$ is computed during the precomputation. Two points in affine λ-coordinates are added in this step. The point addition formula for adding $P = (x_P, \lambda_P)$ and $Q = (x_Q, \lambda_Q)$ is shown in (10). It returns the point $P+Q = (X_{P+Q}, L_{P+Q}, Z_{P+Q})$. An operation sequence for computing (10) is shown in Algorithm 3 in the Appendix.

$$\begin{aligned} A &= \lambda_P + \lambda_Q, \\ B &= (x_P + x_Q)^2, \\ X_{P+Q} &= A \cdot x_P \cdot x_Q \cdot A, \\ Z_{P+Q} &= A \cdot B, \\ L_{P+Q} &= (A \cdot x_Q + B)^2 + A \cdot B \cdot (\lambda_P + 1) \end{aligned} \tag{10}$$

The point doubling formula for a point in projective λ-coordinates $Q = (X_Q, L_Q, Z_Q)$ is shown in (11). It returns the point $2Q = (X_{2Q}, L_{2Q}, Z_{2Q})$. An operation sequence for computing (11) is shown in Algorithm 4 in the Appendix.

$$\begin{aligned} T &= {L_Q}^2 + L_Q \cdot Z_Q + a \cdot {Z_Q}^2, \\ X_{2Q} &= T^2, \\ Z_{2Q} &= T \cdot {Z_Q}^2, \\ L_{2Q} &= (X_Q \cdot Z_Q)^2 + X_{2Q} + T \cdot (L_Q \cdot Z_Q) + Z_{2Q} \end{aligned} \tag{11}$$

The efficiency of combined point doubling and addition is one of the reasons why λ-coordinates give faster results [37]. Computing them separately takes $13\mathbf{M} + 6\mathbf{S}$ in total, whereas the combined point doubling and addition requires only $11\mathbf{M} + 6\mathbf{S}$. Therefore, combined point doubling and addition saves 2 multiplications in $\mathbb{F}_{2^{254}}$. Since the double scalar multiplication with Shamir's trick requires significantly more combined point doublings and additions (for 75 % of the columns, on average) than point doublings (for 25 % of the columns), this trick significantly reduces the overall execution time. A formula for combined point doubling and addition with the inputs $Q = (X_Q, L_Q, Z_Q)$ and $P = (x_P, \lambda_P)$ is shown in (12). It returns the point $2Q + P = (X_{2Q+P}, L_{2Q+P}, Z_{2Q+P})$. An operation sequence for computing (12) is shown in Algorithm 5 in the Appendix.

$$\begin{aligned} A &= {X_Q}^2 \cdot {Z_Q}^2 + T \cdot ({L_Q}^2 + (a + 1 + \lambda_P) \cdot {Z_Q}^2), \\ B &= (x_P \cdot {Z_Q}^2 + T)^2, \\ X_{2Q+P} &= (x_P \cdot {Z_Q}^2) \cdot A^2, \\ Z_{2Q+P} &= A \cdot B \cdot {Z_Q}^2, \\ L_{2Q+P} &= T \cdot (A + B)^2 + (\lambda_P + 1) \cdot Z_{2Q+P} \end{aligned} \tag{12}$$

3.3 Quadratic Extension FSM

Operations in the quadratic extension field (multiplication, constant multiplication, squaring, addition and inversion in $\mathbb{F}_{2^{254}}$) and the endomorphism $\psi(P)$ are implemented by the quadratic operations FSM. The quadratic operations FSM drives the MALU and generates the addresses of input operands of the MALU and generates the output write address for the BRAM. Operation costs of multiplication, squaring, addition and inversion are given in Section 2.2. In order to save time, constant multiplication with the curve parameter a is optimized. As discussed in Section 2.4, $a = u$ for this curve (i.e., $a_0 = 0$ and $a_1 = 1$). Therefore, multiplication $c = a \cdot b$ simplifies to $c = u \cdot (b_0 + b_1 \cdot u) = b_0 \cdot u + b_1 \cdot u^2 \equiv b_1 + (b_0 + b_1) \cdot u \pmod{g(u)}$. Therefore, multiplication by a only requires the swapping of the two halves followed by an addition in $\mathbb{F}_{2^{127}}$. I.e., the cost is only $\mathbf{a}$ and we save $4\mathbf{m} + 2\mathbf{a}$ compared to a general multiplication. The constant multiplication is used once in every iteration of the double scalar multiplication because it is needed once both in point doubling and combined point doubling and addition. The endomorphism ψ is performed as shown in (7) and it takes three modular additions in $\mathbb{F}_{2^{127}}$. The Itoh-Tsujii inversion algorithm is implemented for coordinate conversions. Itoh-Tsujii inversion uses an addition chain of length 9 as given in [37].

3.4 Modular Arithmetic Logic Unit (MALU)

Operations in $\mathbb{F}_{2^{127}}$ (addition, multiplication and squaring) are performed by the MALU which is imported from [39]. The MALU in [39] supports multiplication and addition in $\mathbb{F}_{2^m}$. The MALU is basically a most-significant digit first digit-serial modular multiplier over $\mathbb{F}_{2^m}$ with digit size d. It is adjustable for different irreducible polynomials and digit sizes, which are fixed at the time of implementation. The support for additions is added to the multiplier architecture with a very low cost by utilizing resource sharing. The latency of multiplication is $\lceil \frac{m}{d} \rceil + 2$ clock cycles, where m is the bit size of the operands which are elements in $\mathbb{F}_{2^m}$. The digit size of the MALU is chosen to be $d = 16$ for this implementation as it provides a good tradeoff between area and latency. Therefore, one multiplication in $\mathbb{F}_{2^{127}}$ takes $\lceil \frac{127}{16} \rceil + 2 = 10$ clock cycles. Addition takes 3 clock cycles. The MALU in [39] does not include a dedicated squaring circuitry and, therefore, squaring takes the same time as multiplication. Squaring in $\mathbb{F}_{2^m}$ is a very simple operation and support for it can be added with a very small overhead. Because the point operations involve several squarings, adding the support for squaring provides a significant speedup. Hence, we extended the MALU from [39] with a dedicated squarer. One squaring takes only 3 clock cycles which makes it as fast as an addition. For all field operations, one clock cycle is consumed by loading the inputs and another one goes to a one-stage pipeline for the inputs of the MALU in order to shorten the critical path. The remaining cycles are the actual operation time of the MALU.

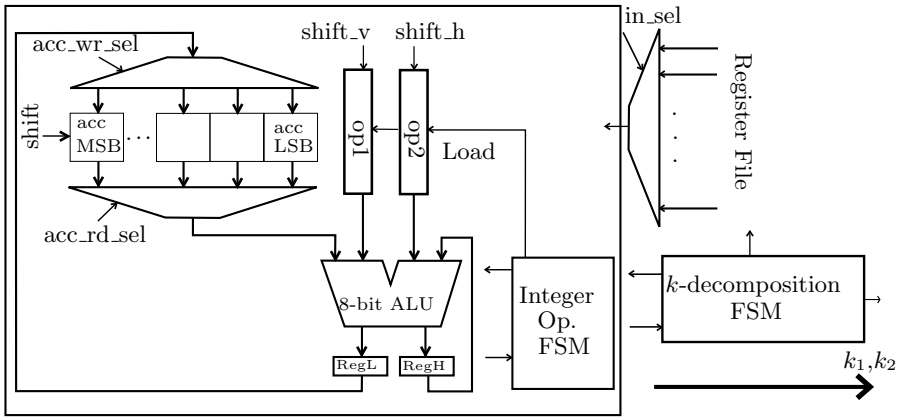

Fig. 3. The architecture for computing the decomposition of k

3.5 Scalar Decomposition

The scalar decomposition is computed in the scalar-decomposition module which contains an 8-bit ALU that performs integer addition, subtraction and multiplication operations. An FSM splits operations with large operands into 8-bit operations and another FSM controls the execution of the decomposition equations given in (8) and (9). Scalar splitting is performed as described in [13]. After rearranging the equations, maximum operand sizes reduce to 192 bits for addition and subtraction and 128 bits for multiplication. These large operands are processed in 8-bit pieces by the ALU. The integer multiplication is implemented using Algorithm 5.1 given in [38]. Since the decomposition is executed in the beginning, it uses free space in the BRAM. The integer multiplication uses one of the dedicated multipliers in the FPGA (one DSP block). The architecture of the scalar-decomposition module is shown in Figure 3.

3.6 Optimizations

Point operations are optimized for minimum execution time and the temporary variable count is optimized based on this minimum timing constraint. Since the most complex equations stand for the combined point doubling and addition, it has the largest temporary variable count, which is 7 in our case. Temporary variables are stored in the BRAM excluding one of them ($R3$) which is in the register file. The base point P, $\psi(P)$, $P + \psi(P)$, and curve constants are also stored in the BRAM. When the MALU is processing, one of the operands is stored in the register file. Also the output from the MALU is stored into the registers (two 127-bit registers) so that it can be used for the next operation, which is usually the case. Another 127-bit register is used for storing temporary results of multiplication and inversion in $\mathbb{F}_{2^{254}}$. Therefore, there are in total three 127-bit registers in the register file. The latencies of point operations are given

Table 1. Point operation latencies (clock cycles)

Operation	**M** $3\mathbf{m}+4\mathbf{a}$	**S** $2\mathbf{s}+\mathbf{a}$	**A** $2\mathbf{a}$	Estimated cycles	Actual cycles
Point doubling and addition	11	6	9	$33\mathbf{m}+12\mathbf{s}+68\mathbf{a}=531$	565
Point doubling	5	4	5	$15\mathbf{m}+8\mathbf{s}+34\mathbf{a}=237$	258
Point addition	5	2	5	$15\mathbf{m}+4\mathbf{s}+32\mathbf{a}=258$	278

in Table 1. The differences between estimated and actual cycle counts arise from access times to the BRAM between operations.

The control logic is pipelined in order to shorten the critical path and to ensure that the critical path is in the processing units (i.e., in the MALU or the 8-bit ALU). Output multiplexers of the register file are pipelined before feeding the data into the MALU. Although this pipeline increases the execution time of every field operation by one clock cycle, the maximum frequency increases significantly. The slice count does not increase significantly because the slice based structure of FPGAs allows to use the flip-flops of a slice essentially for free if the LUT of the slice is already used. Other pipelined paths are address and control signals of the quadratic operations FSM that are controlling the register file and the BRAM. The 8-bit ALU also includes a one-stage pipeline. We experimented with different word sizes for the ALU and selected 8 bits because it was the optimal choice considering the critical path and area.

4 Results and Comparison

The architecture of the coprocessor was described in VHDL. This code was compiled with the Xilinx ISE 12.2 tool for Virtex-5 XC5VLX85-3FF676, Virtex-4 XC4VLX200-11FF1513 and Spartan-6 XC6SLX45T-3FGG484 FPGAs. Simulations for verifying the functionality of the implementations were performed with ModelSim software. Detailed area results after placement & routing are given in Table 2 for Virtex-5 XC5VLX85-3FF676. We compiled the design also for Virtex-4 and Spartan-6 FPGAs. Virtex-4 was selected in order to provide a fair comparison with previous works and Spartan-6 shows the performance of our architecture on a low-cost FPGA, which is commonly used in security-critical embedded systems.

The maximum clock frequency after synthesis is 274.982 MHz for the Virtex-5 implementation. This frequency was added as a design constraint for the place & route and it was able to meet this timing goal. One scalar multiplication takes on average around 61,300 clock cycles, but the exact latency depends on the scalar (the length and the number of nonzero columns in the scalar matrix). Therefore, the total time for an entire scalar multiplication is around 0.223 ms. Precomputation, scalar decomposition and coordinate transformations are also included in the given execution time.

There are many FPGA implementations of elliptic curve scalar multiplications over binary fields. However, most of them do not utilize curves with fast en-

Table 2. Hardware area cost on a Virtex-5 XC5VLX85-3FF676

Module	FF	LUT	BRAM	DSP48
MALU	399	1263	-	-
Scalar-decomposition	667	1025	-	1
Register File	381	1018	-	-
Top FSM	696	911	-	-
Point Op. FSM	44	171	-	-
Quad Op. FSM	42	142	-	-
Bus Controller	10	16	-	-
RAM	-	-	6	-
Total	2239	4546	6	1

domorphisms. The most popular curve has been the NIST B-163 curve from [35]. It provides a lower 80-bit security level, but because finite field arithmetic in $\mathbb{F}_{2^{163}}$ cannot be decomposed into a smaller field, the complexity of FFPU is actually larger. However, the latency of computing (1) on curves with lower security levels is expected to be significantly shorter because fewer operations are computed. These differences should be considered when the designs are compared.

A summary of related work and our implementations is shown in Table 3. The design proposed by Sinha Roy et al. in [41] uses LD coordinates for a binary field EC implementation. They implemented a Karatsuba hybrid multiplier for $\mathbb{F}_{2^{163}}$. The design by Ansari and Hasan in [1] uses a Montgomery multiplier and represent points in projective coordinates. The work by Liu et al. in [24] is a very fast implementation for NIST B-163. It computes a scalar multiplication in only $9\,\mu$s. However, it occupies a huge area due to the extensive use of parallelism in the processor. The implementation presented by Lutz and Hasan in [28] computes a scalar multiplication on NIST K-163, which is a Koblitz curve over $\mathbb{F}_{2^{163}}$, and represents points in LD coordinates. A scalable design that implements all NIST Koblitz curves was proposed by Loi and Ko in [26]. In [42], various NIST curve implementations were proposed by Sutter et al. Their architecture uses three multipliers and, therefore, it gives fast results at the expense of area. To the best of our knowledge, the only other implementation that uses the GLS technique is an implementation described by Azarderakhsh and Karabina in [2], but they do not use λ-coordinates. The implementation provides results on the same security level but it targets high speed at the expense of area which makes comparisons difficult. However, we expect that also their implementation would benefit from the use of λ-coordinates through the reduction of the number of field operations. When our results are compared to the other works in Table 3, it must be borne in mind that our implementation offers about 128 bits of security whereas the ones for B/K-163 and B/K-233 curves offer only about 80 or 112 bits of security, respectively. The security level has significant effects on both the hardware complexity and the total execution time.

As can be seen from Table 3, our design is faster than some related works, although it offers more security. Also, the area consumption of our design is very low compared to other FPGA implementations available in the literature.

Table 3. Comparison with related works

Design	Platform	Time (μs)	Slice	LUT	FF	Curve	Coordinate System
Sinha Roy [41]	Virtex 5	9.5	3513	10195	-	B-163	LD
Ansari [1]	Virtex 2	41	-	8300	1100	B-163	Projective
Liu [24]	Virtex 4	9	10413	-	-	B-163	Projective
Lutz [28]	Virtex 2	446	-	7362	1930	K-163	LD
Loi [26]	Virtex 4	603	2431	3815	1219	K-233	Mixed
Sutter [42]	Virtex 5	19.89	6487	22340	-	B-233	Projective
Chelton [11]	Virtex 4	19.55	16209	-	-	B-163	Projective
Azarderakhsh [2]	Virtex 4	16.85	12043	-	-	GLS-254	Projective
This work	Virtex 4	317	3985	7112	2247	GLS-254	λ
This work	Virtex 5	223	1552	4546	2239	GLS-254	λ
This work	Spartan-6	370	1546	4983	2315	GLS-254	λ

Hence, our design is suitable for embedded systems that require fast scalar multiplications with a small footprint. In such applications, low-cost FPGAs are more feasible implementation platforms than high-end FPGAs such as the Virtex family FPGAs. Hence, we also give implementation result for a low-cost FPGA from the Spartan-6 family in Table 3. Our results show that the new technique of using λ-coordinates with the GLS endomorphism introduced in [37] results in a fast and compact ECC coprocessor that can be used in various security-critical embedded systems. Note that a slice consists of several LUTs and FFs. In Table 3, we give both the number of slices and the number of LUTs and FFs in order to compare to other implementations that only give one of both.

5 Conclusion

We presented a fast and compact FPGA implementation of ECC with a 128-bit security level. Our implementation uses many of the optimizations used in the software implementation presented in [37] by Oliveira et al. To the best of our knowledge, it is the first FPGA implementation that utilizes λ-coordinates and the GLS decomposition.

We demonstrated that these techniques do not only offer significant advantages in software implementations but that improvements can be obtained also for hardware implementations. An especially significant advantage in the case of hardware is that finite field arithmetic is decomposed so that it can be performed in a small field, where m is of the size of the security level. This results in small areas and high maximum clock frequencies. In the case of many other curves (e.g., the NIST curves), field arithmetic is performed in fields with sizes of twice the security level. We showed also that the GLS decompositions can be computed efficiently in hardware, although extra hardware is required for the integer operations that are not natively supported by field arithmetic units (such as the MALU that we used in this paper). This extra cost is small compared to the benefits that can be obtained from the faster scalar multiplication.

The target applications for our implementation are various security-critical embedded systems that require high-security public-key cryptography. Hence, primary concerns are small area and good speed-area ratio and not necessarily the lowest computation time or highest throughput (scalar multiplications per second). In fact, the highly optimized software implementation reported in [37] requires only 47,900 clock cycles per scalar multiplication on Intel Sandy Bridge processors which run at significantly higher clock frequencies (e.g., 3.4 GHz) compared to our implementation (274 MHz) and, therefore, it outperforms our FPGA implementation. However, this kind of performance is completely out of the reach of embedded software implementations. For example, a single scalar multiplication requires some tens of milliseconds on a 32-bit processor [12] and more than 0.5 s on an 8-bit microcontroller [23, 25]. Thus, our implementation provides major performance advantages compared to software implementations typically used in security-critical embedded systems.

We foresee several directions for future research on this topic. A careful parameter space exploration for our architecture would allow finding the optimum parameters (e.g., the digit-size of the MALU) which would allow finding the optimum speed-area ratio. Countermeasures against side-channel attacks need to be implemented in order to use our implementation in applications where side-channel attacks are a threat. In particular, timing and operation patterns need to be constant in order to thwart timing attacks and simple power analysis or electromagnetic attacks. Further speedups can be obtained by using precomputations (e.g., w-NAF) and by using the two-core parallelization technique based on double-and-add and halve-and-add from [37]. Further increase in throughput can be achieved by instantiating multiple cores in a single FPGA. For instance, we extrapolated that about 25 parallel cores would fit into the largest Virtex-5 FPGA (filled up to 80 %). Hence, our implementation can be valuable also for accelerating cryptographic computations.

Acknowledgments

K. Järvinen is an FWO Pegasus Marie Curie Fellow funded by Fonds Wetenschappelijk Onderzoek—the Research Foundation of Flanders. Further, this work was supported in part by the Research Council KU Leuven (C16/15/058) and IOF project EDA-DSE (HB/13/020).

References

1. Ansari, B., Hasan, M.A.: High-performance architecture of elliptic curve scalar multiplication. IEEE Transactions on Computers 57(11), 1443–1453 (2008)
2. Azarderakhsh, R., Karabina, K.: A new double point multiplication method and its implementation on binary elliptic curves with endomorphisms. Tech. Rep. CACR 2012-24, University of Waterloo, Centre for Applied Cryptographic Research (2012)
3. Azarderakhsh, R., Reyhani-Masoleh, A.: Efficient FPGA implementations of point multiplication on binary Edwards and generalized Hessian curves using Gaussian normal basis. IEEE Transactions on VLSI Systems 20(8), 1453–1466 (2012)

4. Azarderakhsh, R., Reyhani-Masoleh, A.: High-performance implementation of point multiplication on Koblitz curves. IEEE Transactions on Circuits and Systems II: Express Briefs 60(1), 41–45 (2013)
5. Baldwin, B., Moloney, R., Byrne, A., McGuire, G., Marnane, W.P.: A hardware analysis of twisted Edwards curves for an elliptic curve cryptosystem. In: Reconfigurable Computing: Architectures, Tools and Applications, pp. 355–361. Springer (2009)
6. Batina, L., Hogenboom, J., Mentens, N., Moelans, J., Vliegen, J.: Side-channel evaluation of FPGA implementations of binary Edwards curves. In: Proceedings of the 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2010). pp. 1248–1251. IEEE (2010)
7. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Public Key Cryptography—PKC 2006, LNCS, vol. 3958, pp. 207–228. Springer (2006)
8. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Progress in Cryptology—AFRICACRYPT 2008, LNCS, vol. 5023, pp. 389–405. Springer (2008)
9. Bernstein, D.J., Lange, T., Farashahi, R.R.: Binary Edwards curves. In: Cryptographic Hardware and Embedded Systems—CHES 2008, LNCS, vol. 5154, pp. 244–265. Springer (2008)
10. Bos, J.W., Costello, C., Longa, P., Naehrig, M.: Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130 (2014), http://eprint.iacr.org/
11. Chelton, W.N., Benaissa, M.: Fast elliptic curve cryptography on FPGA. IEEE Transactions on VLSI Systems 16(2), 198–205 (2008)
12. De Clercq, R., Uhsadel, L., Van Herrewege, A., Verbauwhede, I.: Ultra low-power implementation of ECC on the ARM Cortex-M0+. In: Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference (DAC 2014). pp. 1–6. ACM (2014)
13. Galbraith, S.D., Lin, X., Scott, M.: Endomorphisms for faster elliptic curve cryptography on a large class of curves. Journal of Cryptology 24(3), 446–469 (2011)
14. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphisms. In: Advances in Cryptology—CRYPTO 2001. LNCS, vol. 2139, pp. 190–200. Springer (2001)
15. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and RSA on 8-bit cpus. In: Cryptographic Hardware and Embedded Systems—CHES 2004, LNCS, vol. 3156, pp. 119–132. Springer (2004)
16. Hamilton, M., Marnane, W.P.: FPGA implementation of an elliptic curve processor using the GLV method. In: Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'09). pp. 249–254. IEEE (2009)
17. Hankerson, D., Karabina, K., Menezes, A.: Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. IEEE Transactions on Computers 58(10), 1411–1420 (2009)
18. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer (2004)
19. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Information and Computation 78(3), 171–177 (1988)
20. Järvinen, K.: Optimized FPGA-based elliptic curve cryptography processor for high-speed applications. Integration, the VLSI journal 44(4), 270–279 (2011)
21. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)

22. Koblitz, N.: CM-curves with good cryptographic properties. In: Advances in Cryptology — CRYPTO '91. LNCS, vol. 576, pp. 279–287. Springer (1991)
23. Liu, A., Ning, P.: TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In: Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'08). pp. 245–256. IEEE (2008)
24. Liu, S., Ju, L., Cai, X., Jia, Z., Zhang, Z.: High performance FPGA implementation of elliptic curve cryptography over binary fields. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on. pp. 148–155. IEEE (2014)
25. Liu, Z., Seo, H., Großschädl, J., Kim, H.: Efficient implementation of NIST-compliant elliptic curve cryptography for sensor nodes. In: Proceedings of the 15th International Conference on Information and Communications Security (ICICS 2013), LNCS, vol. 8233, pp. 302–317. Springer International Publishing (2013)
26. Loi, K.C., Ko, S.B.: High performance scalable elliptic curve cryptosystem processor for Koblitz curves. Microprocessors and Microsystems 37(4), 394–406 (2013)
27. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in $GF(2^n)$. In: Selected Areas in Cryptography — SAC'98. LNCS, vol. 1556, pp. 201–212. Springer (1999)
28. Lutz, J., Hasan, A.: High performance FPGA based elliptic curve cryptographic coprocessor. In: Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on. vol. 2, pp. 486–492. IEEE (2004)
29. Menezes, A.J., Okamoto, T., Vanstone, S.A.: Reducing elliptic curve logarithms to logarithms in a finite field. IEEE Transactions on Information Theory 39(5), 1639–1646 (1993)
30. Menezes, A.J., Vanstone, S.A.: Elliptic curve cryptosystems and their implementation. Journal of Cryptology 6(4), 209–224 (1993)
31. Miller, V.S.: Use of elliptic curves in cryptography. In: Advances in Cryptology — CRYPTO '85. LNCS, vol. 218, pp. 417–426. Springer (1986)
32. National Institute of Standards and Technology (NIST): Advanced encryption standard (AES). Federal Information Processing Standard, FIPS PUB 197 (2001)
33. National Institute of Standards and Technology (NIST): Recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A Transitions (2011)
34. National Institute of Standards and Technology (NIST): Recommendation for key management — Part 1: General. NIST Special Publication 800-57 (2012)
35. National Institute of Standards and Technology (NIST): Digital signature standard (DSS). Federal Information Processing Standard, FIPS PUB 186-4 (2013)
36. National Institute of Standards and Technology (NIST): NIST workshop on elliptic curve cryptography standards. Call for papers (2015), http://www.nist.gov/itl/csd/ct/upload/CFP-Elliptic-Curve-Crypto-June2015.pdf
37. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Lambda coordinates for binary elliptic curves. In: Cryptographic Hardware and Embedded Systems — CHES 2013. LNCS, vol. 8086, pp. 311–330. Springer (2013)
38. Rodríguez-Henríquez, F., Saqib, N.A., Perez, A.D., Koç, C.K.: Cryptographic algorithms on reconfigurable hardware. Springer Science & Business Media (2007)
39. Sakiyama, K., Batina, L., Mentens, N., Preneel, B., Verbauwhede, I.: Small-footprint ALU for public-key processors for pervasive security. In: Workshop on RFID Security — RFIDSec 2006 (2006)
40. Sasdrich, P., Güneysu, T.: Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In: Reconfigurable Computing: Architectures, Tools, and Applications, LNCS, vol. 8405, pp. 25–36. Springer International Publishing (2014)

41. Sinha Roy, S., Rebeiro, C., Mukhopadhyay, D.: Theoretical modeling of elliptic curve scalar multiplier on LUT-based FPGAs for area and speed. IEEE Transactions on VLSI Systems 21(5), 901–909 (2013)
42. Sutter, G.D., Deschamps, J., Imaña, J.L.: Efficient elliptic curve point multiplication using digit-serial binary field operations. IEEE Transactions on Industrial Electronics 60(1), 217–225 (2013)
43. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication. In: Cryptographic Hardware and Embedded Systems—CHES 2011, LNCS, vol. 6917, pp. 108–123. Springer (2011)

A Algorithms for Point Operations

Algorithm 3 Point addition $P + Q$

1. $R3 \leftarrow x_P + x_Q$
2. $RL_Q \leftarrow B = R3^2$
3. $R3, R0 \leftarrow A = \lambda_P \cdot \lambda_Q$
4. $R3, R1 \leftarrow R3 \cdot x_Q$
5. $R3 \leftarrow R3 \cdot x_P$
6. $RX_Q \leftarrow R3 \cdot R0$
7. $R3 \leftarrow R1 + B$
8. $R1 \leftarrow R3^2$
9. $RZ_Q \leftarrow Z_{P+Q} = R0 \cdot RL_Q$
10. $R3 \leftarrow \lambda_P + 1$
11. $R3 \leftarrow RZ_Q \cdot R3$
12. $RL_Q \leftarrow L_{P+Q} = R1 + R3$

Algorithm 4 Point doubling $2Q$

1. $R0 \leftarrow {L_Q}^2$
2. $R3, RL_Q \leftarrow L_Q \cdot Z_Q$
3. $R0 \leftarrow R0 + R3$
4. $R1 \leftarrow {Z_Q}^2$
5. $R3 \leftarrow a \cdot R1$
6. $R0 \leftarrow T = R0 + R3$
7. $R3 \leftarrow X_Q \cdot Z_Q$
8. $R2 \leftarrow R3^2$
9. $RX_Q \leftarrow X_{2Q} = R0^2$
10. $RZ_Q \leftarrow Z_{2Q} = R0 \cdot R1$
11. $R3 \leftarrow R0 \cdot RL_Q$
12. $R3 \leftarrow R3 + R2$
13. $R3 \leftarrow R3 + RX_Q$
14. $RL_Q \leftarrow L_{2Q} = R3 + RZ_Q$

Algorithm 5 Combined point doubling and addition $Q = 2Q + P$

1. $R0 \leftarrow {L_Q}^2$
2. $R3 \leftarrow L_Q \cdot Z_Q$
3. $RL_Q \leftarrow R0 + R3$
4. $R3, RZ_Q \leftarrow Z_Q^2$
5. $R3 \leftarrow a \cdot R3$
6. $R1 \leftarrow T = RL_Q + R3$
7. $R3, RL_Q \leftarrow 1 + \lambda_P$
8. $R3 \leftarrow a + R3$
9. $R3 \leftarrow R3 \cdot RZ_Q$
10. $R3, R0 \leftarrow R0 + R3$
11. $R0 \leftarrow R1 \cdot R3$
12. $R3 \leftarrow X_Q^2$
13. $R3 \leftarrow R3 \cdot RZ_Q$
14. $R0 \leftarrow A = R0 + R3$
15. $R3, RX_Q \leftarrow x_P \cdot RZ_Q$
16. $R3 \leftarrow R1 + R3$
17. $R2 \leftarrow B = R3^2$
18. $R3 \leftarrow R0^2$
19. $RX_Q \leftarrow X_{2Q+P} = R3 \cdot RX_Q$
20. $R3 \leftarrow R0 \cdot R2$
21. $R3, RZ_Q \leftarrow Z_{2Q+P} = R3 \cdot RZ_Q$
22. $RL_Q \leftarrow RL_Q \cdot R3$
23. $R3 \leftarrow R0 + R2$
24. $R3 \leftarrow R3^2$
25. $R3 \leftarrow R1 \cdot R3$
26. $RL_Q \leftarrow L_{2Q+P} = R3 \cdot RL_Q$

The formulas used for deriving the algorithms are available in Explicit-Formulas Database (https://hyperelliptic.org/EFD/g12o/auto-shortw-lambda.html). It also includes verification scripts (Sage).