Lightweight Coprocessor for Koblitz Curves: 283-Bit ECC Including Scalar Conversion with only 4300 Gates

Sujoy Sinha Roy(✉), Kimmo Järvinen, and Ingrid Verbauwhede

KU Leuven ESAT/COSIC and iMinds, Kasteelpark Arenberg 10 bus 2452, 3001 Leuven-Heverlee, Belgium
{sujoy.sinharoy,kimmo.jarvinen,ingrid.verbauwhede}@esat.kuleuven.be

Abstract. We propose a lightweight coprocessor for 16-bit microcontrollers that implements high security elliptic curve cryptography. It uses a 283-bit Koblitz curve and offers 140-bit security. Koblitz curves offer fast point multiplications if the scalars are given as specific τ-adic expansions, which results in a need for conversions between integers and τ-adic expansions. We propose the first lightweight variant of the conversion algorithm and, by using it, introduce the first lightweight implementation of Koblitz curves that includes the scalar conversion. We also include countermeasures against side-channel attacks making the coprocessor the first lightweight coprocessor for Koblitz curves that includes a set of countermeasures against timing attacks, SPA, DPA and safe-error fault attacks. When the coprocessor is synthesized for 130 nm CMOS, it has an area of only 4,323 GE. When clocked at 16 MHz, it computes one 283-bit point multiplication in 98 ms with a power consumption of 97.70 μW, thus, consuming 9.56 μJ of energy.

Keywords: Elliptic curve cryptography · Koblitz curves · Lightweight implementation · Side-channel resistance

1 Introduction

Elliptic curve cryptography (ECC) is one of the prime candidates for bringing public-key cryptography to applications with strict constraints on implementation resources such as power, energy, circuit area, memory, etc. Lightweight applications that require strong public-key cryptography include, e.g., wireless sensor network nodes, RFID tags, medical implants, and smart cards. Such applications will have a central role in actualizing concepts such as the Internet of Things and, hence, providing strong cryptography with low resources has been an extremely active research field in the recent years. As a result of this research line, we have several proposals for efficient lightweight implementations of ECC. These proposals focus predominately on 163-bit elliptic curves which provide medium security level of about 80 bits [4–6,15,24,26,34,42,43]. We provide a coprocessor architecture that implements ECC using a high security 283-bit Koblitz curve and includes countermeasures against side-channel attacks.

© International Association for Cryptologic Research 2015
T. Güneysu and H. Handschuh (Eds.): CHES 2015, LNCS 9293, pp. 102–122, 2015.
DOI: 10.1007/978-3-662-48324-4_6

Koblitz curves [23] are a special class of elliptic curves which enable very efficient point multiplications and, therefore, they are an attractive alternative also for lightweight implementations. However, these efficiency gains can be exploited only by representing scalars as specific τ-adic expansions. Most cryptosystems require the scalar also as an integer (see, e.g., ECDSA [31]). Therefore, cryptosystems utilizing Koblitz curves need both the integer and τ-adic representations of the scalar, which results in a need for conversions between the two domains. This is not a major problem in applications which have sufficient resources because fast methods for on-the-fly scalar conversion are available [8,37]. Consequently, very fast ECC implementations using Koblitz curves have been presented for both software [39] and hardware [18]. For lightweight implementations, however, the extra overhead introduced by these conversions has so far prevented efforts to use Koblitz curves in lightweight implementations. A recent paper [4] showed that Koblitz curves result in a very efficient lightweight implementation if τ-adic expansions are already available but the fact that the conversion is not included seriously limits possible applications of the implementation. An alternative approach was provided in a very recent paper [20] which provides a solution that delegates conversions from the lightweight implementation to a powerful server. However, this solution is not suitable for applications where both communicating parties are lightweight implementations and it also requires minor modifications to the cryptosystems which may hinder its use in some applications. Computing conversions directly in the lightweight implementation would be a better option in many cases and, hence, we focus on that alternative in this paper. All previous hardware implementations of the conversions [1,7,8,19,36] are targeted on high speed which makes them unsuitable for lightweight implementations.

To the best of our knowledge, we present the following novel contributions:

- We present the first lightweight implementation of high security ECC by using a 283-bit Koblitz curve offering roughly 140 bits of security. By high security, we mean security levels exceeding 128 bits (e.g., AES-128). Because security of a cryptosystem utilizing multiple cryptographic algorithms is determined by its weakest algorithm, our implementation is the first lightweight implementation of ECC that can be combined, e.g., with AES-128 without reducing the security level of the entire system.
- We present the first complete lightweight implementation of Koblitz curves that also includes on-the-fly scalar conversion. We achieve this by presenting a lightweight variant of the conversion algorithm from [8] which is optimized for word-serial computations. As mentioned above, the first implementation introduced in [4] does not include the conversion which limits the possible applications of the implementation. All conversion algorithms and architectures available in the literature focus on the speed of the conversion.
- The first lightweight implementation of Koblitz curves [4] does not include any countermeasures against side-channel attacks. We present the first lightweight implementation of Koblitz curves with countermeasures against side-channel attacks such as simple power analysis (SPA), differential power analysis (DPA), timing attacks, and safe-error fault attacks.

The paper is structured as follows. In Sect. 2, we provide a brief background on ECC and Koblitz curves. Then in Sects. 3 and 4, we describe our scalar conversion and point multiplication techniques. Our lightweight coprocessor architecture is presented in Sect. 5. We provide synthesis results in 130 nm CMOS and comparisons to other works in Sect. 6. We end with conclusions in Sect. 7.

2 Preliminaries

The use of elliptic curves for cryptography was independently proposed by Victor Miller [29] and Neal Koblitz [22] in the mid-1980 s. Points that satisfy the equation of an elliptic curve form an additive Abelian group E together with a special point $\mathcal{O}$, which is the zero element of the group. Elliptic curves over finite fields $\mathbb{F}_q$ are used in cryptography and we focus on elliptic curves over binary fields $\mathbb{F}_{2^m}$ (finite fields over characteristic two) with polynomial basis. Let $P_1, P_2 \in E$. The group operation $P_1 + P_2$ is called point addition when $P_1 \neq \pm P_2$ and point doubling when $P_1 = P_2$. The fundamental operation of ECC is the elliptic curve point multiplication $Q = kP$, where $k \in \mathbb{Z}$ and $Q, P \in E$.

Point multiplication is computed with a series of point additions and point doublings. The basic approach to compute point multiplications is to use the double-and-add algorithm (also called the binary algorithm) which iterates over the bits of k one at a time and computes a point doubling for every bit and a point addition if the bit is one. Each point operation involves several operations in the underlying finite field. Projective coordinates are typically used for representing points as (X, Y, Z) in order to reduce the number of inversions in $\mathbb{F}_{2^m}$. We use the López-Dahab coordinates [27] and specifically the point addition formulae from [2]. Another option that we considered was to use the λ-coordinates [33] which offer slightly faster point additions. In our case, however, the cost of obtaining the λ-coordinate representation, which includes an inversion in $\mathbb{F}_{2^m}$, overweighs the cheaper point additions.

Koblitz curves introduced by Koblitz in [23] are a special class of elliptic curves defined by the following equation:

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

with $x, y \in \mathbb{F}_{2^m}$ and $a \in \{0, 1\}$. Koblitz curves offer efficient point multiplications because they allow trading computationally expensive point doublings to cheap Frobenius endomorphisms. Many standards use Koblitz curves including NIST FIPS 186-4 [31] which describes the (Elliptic Curve) Digital Signature Standard (ECDSA) and defines five Koblitz curves NIST K-163, K-233, K-283, K-409, and K-571 over the finite fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, and $\mathbb{F}_{2^{571}}$, respectively.

The Frobenius endomorphism for a point $P = (x, y)$ is given by $\phi(P) = (x^2, y^2)$ and, for Koblitz curves, it holds that $\phi(P) \in E$ for all $P \in E$. It can be also shown that $\phi^2(P) - \mu\phi(P) + 2P = \mathcal{O}$ for all $P \in E$, where $\mu = (-1)^{1-a}$ [23]. Consequently, the Frobenius endomorphism can be seen as a multiplication by the complex number $\tau = (\mu + \sqrt{-7})/2$ [23].

Representing the scalar k as a τ-adic expansion $t = \sum_{i=0}^{\ell-1} t_i \tau^i$ allows computing point multiplications with a Frobenius-and-add algorithm, which is similar to the double-and-add algorithm except that point doublings are replaced by Frobenius endomorphisms. Depending on the application, a τ-adic expansion can be found by converting an integer into a τ-adic expansion [8,23,28,37] and/or by finding a random τ-adic expansion directly [23,25]. In the latter case, a conversion in the other direction is typically required because most cryptosystems (e.g., ECDSA [31]) require the scalar as an integer, too. Conversions in either direction can be expensive [8] but once the τ-adic expansion is obtained, the point multiplication is significantly faster, which typically makes Koblitz curves more efficient than other standardized elliptic curves. So far, no efficient lightweight implementations of these conversions exist ruling Koblitz curves out of the domain of lightweight cryptography.

Because the negative of a point is given simply as $-P = (x, x+y)$, the cost of point subtraction is practically equal to the cost of point addition and significant performance improvements can be obtained by using signed-bit representations for the scalar. In that case, a point addition is computed if $t_i = +1$ and a point subtraction is computed if $t_i = -1$. The most widely used signed-bit representation for Koblitz curves is the τ-adic nonadjacent form (τNAF) introduced by Solinas in [37] and it has an average density of $1/3$ for nonzero coefficients. Solinas also presented the window τNAF (w-τNAF) that allows even lower densities of $1/(w+1)$ by utilizing precomputations to support an increased set of possible values for coefficients: $t_i \in \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$.

Both τNAF and w-τNAF have the serious downside that they are vulnerable against side-channel attacks because the pattern of point operations depends on the key bits. The basic approach for obtaining resistance against side-channel attacks for ECC is to use Montgomery's ladder [30] which employs a constant pattern of point operations. Unfortunately, Montgomery's ladder is not a viable choice for Koblitz curves because then all benefits of cheap Frobenius endomorphisms are lost. Certain options (e.g., by using dummy operations) have been proposed in [14]. In this paper, we reuse the idea of using a zero-free τ-adic representation [32,41], that contains only nonzero digits, i.e., $t_i \in \{-1, +1\}$. When this representation is scanned with windows of size $w \geq 2$, the resulting point multiplication algorithm is both efficient and secure against many side-channel attacks because it employs a constant pattern of operations [32,41].

3 Koblitz Curve Scalar Conversion

The zero-free representation for an integer scalar k is found so that k is first reduced to $\rho = b_0 + b_1\tau \equiv k \pmod{\tau^m - 1}$ and the zero-free representation t is generated from the reduced scalar ρ [32,37,41]. The overhead of these conversions is specifically important for lightweight implementations. Another important aspect is resistance against side-channel attacks. In the following, we describe our lightweight and side-channel resistant scalar conversion algorithms. Only SPA countermeasures are required because only one conversion is required per

```
Input: Integer scalar k
Output: Reduced scalar ρ = b_0 + b_1 τ ≡ k (mod τ^m − 1)
1 (a_0, a_1) ← (1, 0), (b_0, b_1) ← (0, 0), (d_0, d_1) ← (k, 0)
2 for i = 0 to m − 1 do
3     u ← d_0[0] ; /* The lsb of d_0, the remainder before division by τ */
4     d_0 ← d_0 − u
5     (b_0, b_1) ← (b_0 + u · a_0, b_1 + u · a_1)
6     (d_0, d_1) ← (d_1 − d_0/2, −d_0/2) ;          /* Division of (d_0, d_1) by τ */
7     (a_0, a_1) ← (−2a_1, a_0 − a_1)
8 ρ = (b_0, b_1) ← (b_0 + d_0, b_1 + d_1)
```

Algorithm 1. Scalar reduction algorithm from [8]

k. The scalar k is typically a nonce but even if it is used multiple times, t can be computed only once and stored.

3.1 Scalar Reduction

We choose the scalar reduction technique called lazy reduction (described as Algorithm 1) from [8]. The algorithm reduces an integer scalar by repeatedly dividing it by τ for m times. This division can be implemented with shifts, additions, and subtractions. This makes the scalar reduction algorithm [8] attractive for lightweight implementations. However, the only known hardware implementations of this algorithm [8] and its speed-optimized versions [1,36] use full-precision integer arithmetic and parallelism to minimize cycle count. Hence the reported architectures consume large areas and are thus not suitable for lightweight implementations. We observe that the original lazy reduction algorithm [8] can also be implemented in a word-serial fashion to reduce area requirements but such a change in the design decision increases cycle count. To reduce the number of cycles, we optimize the computational steps of Algorithm 1. Further, we investigate side-channel vulnerability of the algorithm and propose lightweight countermeasures against SPA.

Computational Optimization. In lines 6 and 7 of Algorithm 1, computations of d_1 and a_0 require subtractions from zero. In a word-serial architecture with only one adder/subtracter circuit, they consume nearly 33 % of the cycles of the scalar reduction. We use the iterative property of Algorithm 1 and eliminate these two subtractions by replacing lines 6 and 7 with the following ones:

$$\begin{aligned}(d_0, d_1) &\leftarrow (d_0/2 - d_1, d_0/2)\\ (a_0, a_1) &\leftarrow (2a_1, a_1 - a_0)\end{aligned} \quad (2)$$

However with this modification, (a_0, a_1) and (d_0, d_1) have a wrong sign after every odd number of iterations of the for-loop in Algorithm 1. It may appear that this wrong sign could affect correctness of (b_0, b_1) in line 5. Since the remainder u (in

line 3) is generated from d_0 instead of the correct value $-d_0$, a wrong sign is also assigned to u. Hence, the multiplications $u \cdot a_0$ and $u \cdot a_1$ in line 5 are always correct, and the computation of (b_0, b_1) remains unaffected of the wrong signs.

After completion of the for-loop, the sign of (d_0, d_1) is wrong as m is an odd integer for secure fields. Hence, the correct value of the reduced scalar should be computed as $\rho \leftarrow (b_0 - d_0, b_1 - d_1)$.

Protection Against SPA. In line 5 of Algorithm 1, computation of new (b_0, b_1) depends on the remainder bit (u) generated from d_0 which is initialized to k. Multi-precision additions are performed when $u = 1$; whereas no addition is required when u is zero. A side-channel attacker can detect this conditional computation and can use, e.g., the techniques from [8] to reconstruct the secret key from the remainder bits that are generated during the scalar reduction.

One way to protect the scalar reduction from SPA is to perform dummy additions $(b_0', b_1') \leftarrow (b_0 + a_0, b_1 + a_1)$ whenever $u = 0$. However, such countermeasures based on dummy operations require more memory and are vulnerable to fault attacks [11]. We propose a countermeasure inspired by the zero-free τ-adic representations from [32,41]. A zero-free representation is obtained by generating the remainders u from $d = d_0 + d_1\tau$ using a map $\Psi(d) \to u \in \{1, -1\}$ such that $d - u$ is divisible by τ, but additionally not divisible by τ^2 (see Sect. 3.2). We observe that during the scalar reduction (which is basically a division by τ), we can generate the remainder bits u as either 1 or -1 throughout the entire for-loop in Algorithm 1. Because $u \neq 0$, new (b_0, b_1) is always computed in the for-loop and protection against SPA is achieved without dummy operations. The following equation generates u by observing the second lsb of d_0 and lsb of d_1.

$$
\begin{aligned}
&\text{Case 1: If } d_0[1] = 0 \text{ and } d_1[0] = 0, \text{ then } u \leftarrow -1 \\
&\text{Case 2: If } d_0[1] = 1 \text{ and } d_1[0] = 0, \text{ then } u \leftarrow 1 \\
&\text{Case 3: If } d_0[1] = 0 \text{ and } d_1[0] = 1, \text{ then } u \leftarrow 1 \\
&\text{Case 4: If } d_0[1] = 1 \text{ and } d_1[0] = 1, \text{ then } u \leftarrow -1
\end{aligned}
\quad (3)
$$

The above equation takes an odd d_0 and computes u such that the new d_0 after division of $d - u$ by τ is also an odd integer.

Algorithm 2 shows our computationally efficient SPA-resistant scalar reduction algorithm. All operations are performed in a word-serial fashion. Since the remainder generation in (3) requires the input d_0 to be an odd integer, the lsb of d_0 is always set to 1 (in line 3) when the input scalar k is an even integer. In this case, the algorithm computes the reduced scalar of $k + 1$ instead of k and after the completion of the reduction, the reduced scalar should be decremented by one. Algorithm 2 uses a one-bit register e to implement this requirement. The final subtraction in line 10 uses e as a borrow to the adder/subtracter circuit. In the next section, we show that the subtraction $d_0 - u$ in line 6 also leaks information about u and propose a countermeasure that prevents this.

```
    Input: Integer scalar k
    Output: Reduced scalar ρ = b0 + b1τ ≡ k (mod τ^m − 1)
 1  (a0, a1) ← (1, 0), (b0, b1) ← (0, 0), (d0, d1) ← (k, 0)
 2  e ← ¬d0[0] ;                                      /* Set to 1 when d0 is even */
 3  d0[0] ← 1
 4  for i = 0 to m − 1 do
 5  |   u ← Ψ(d0 + d1τ) ;      /* Remainder u ∈ {1, −1}, computed using (3) */
 6  |   d0 ← d0 − u
 7  |   (b0, b1) ← (b0 + u · a0, b1 + u · a1)
 8  |   (d0, d1) ← (d0/2 − d1, d0/2) ;               /* Saves one subtraction */
 9  |   (a0, a1) ← (2a1, a1 − a0) ;                  /* Saves one subtraction */
10  ρ = (b0, b1) ← (b0 − d0 − e, b1 − d1) ; /* Subtraction instead of addition */
```

Algorithm 2. SPA-resistant scalar reduction

```
    Input: Reduced scalar ρ = b0 + b1τ with b0 odd
    Output: Zero-free τ-adic bits (t_{ℓ−1}, · · · t0)
 1  i ← 0
 2  while |b0| ≠ 1 or b1 ≠ 0 do
 3  |   u ← Ψ(b0 + b1τ) ;                             /* Computed using (3) */
 4  |   b0 ← b0 − u
 5  |   (b0, b1) ← (b1 − b0/2, −b0/2)
 6  |   ti ← u
 7  |   i ← i + 1
 8  ti ← b0
```

Algorithm 3. Computation of zero-free τ-adic representation [32]

3.2 Computation of τ-adic Representation

For side-channel attack resistant point multiplication, we use the zero-free τ-adic representation proposed in [32,41] and described in Algorithm 3. In this paper, we add the following improvements to the algorithm.

Computational Optimization. Computation of b_1 in line 5 of Algorithm 3 requires subtraction from zero. Similar to Sect. 3.1 this subtraction can be avoided by computing $(b_0, b_1) \leftarrow (b_0/2 - b_1, b_0/2)$. With this modification, the sign of (b_0, b_1) will be wrong after an odd number of iterations. In order to correct this, the sign of t_i should be flipped for odd i (by multiplying it with $(-1)^i$).

Protection Against SPA. Though point multiplications with zero-free representations are resistant against SPA [32], the generation of τ-adic bits (Algorithm 3) is vulnerable to SPA. In line 3 of Algorithm 3, a remainder u is computed as per the four different cases described in (3) and then subtracted from b_0 in line 4. We use the following observations to detect the side-channel vulnerability in this subtraction and to propose a countermeasure against SPA.

1. For Case 1, 2 and 3 in (3), the subtractions of u are equivalent to flipping two (or one) least significant bits of b_0. Hence, actual subtractions are not computed in these cases.
2. For Case 4, subtraction of u from b_0 (i.e. computation of b_0+1) involves carry propagation. Hence, an actual multi-precision subtraction is computed in this case.
3. If any iteration of the while-loop in Algorithm 3 meets Case 4, then the new value of b_1 will be even. Hence, the while-loop will meet either Case 1 or Case 2 in the next iteration.

Based on the differences in computation, a side-channel attacker using SPA can distinguish Case 4 from the other three cases. Hence, the attacker can reveal around 25 % of the bits of a zero-free representation. Moreover, the attacker knows that the following τ-adic bits are biased towards 1 instead of -1 with a probability of 1/3.

We propose a very low-cost countermeasure that skips this special addition $b_0 + 1$ for Case 4 by merging it with the computation of new (b_0, b_1) in Algorithm 3. In line 5, we compute a new b_0 as:

$$b_0 \leftarrow \left(\frac{b_0+1}{2} - b_1\right) = \left(\frac{b_0-1}{2} - \{b_1', 0\}\right). \tag{4}$$

Since b_1 is an odd number for Case 4, we can represent it as $\{b_1', 1\}$ and subtract the least significant bit 1 from $(b_0 + 1)/2$ to get $(b_0 - 1)/2$. Since b_0 is always odd, the computation of $(b_0 - 1)/2$ is just a left-shift of b_0.

The computation of $b_1 \leftarrow (b_0 + 1)/2$ in line 5 of Algorithm 3 involves a carry propagation and thus an actual addition becomes necessary. We solve this problem by computing $b_1 \leftarrow (b_0-1)/2$ instead of the correct value $b_1 \leftarrow (b_0+1)/2$ and remembering the difference (i.e., 1) in a flag register h. Correctness of the τ-adic representation can be maintained by considering this difference in the future computations that use this wrong value of b_1. Now as per observation 3, the next iteration of the while-loop meets either Case 1 or 2. We adjust the previous difference by computing the new b_0 as follows:

$$b_0 \leftarrow \left(\frac{b_0}{2} - (b_1 + h)\right) = \left(\frac{b_0}{2} - b_1 - 1\right). \tag{5}$$

In a hardware architecture, this equation can be computed by setting the borrow input of the adder/subtracter circuit to 1 during the subtraction.

In (6), we show our new map $\Psi'(\cdot)$ that computes a remainder u and a new value h' of the difference flag following the above procedure. We consider $b_1[0] \oplus h$ (instead of $b_1[0]$ as in (3)) because a wrong b_1 is computed in Case 4 and the difference is kept in h.

$$\begin{array}{ll} \text{Case 1: If } b_0[1] = 0 \text{ and } b_1[0] \oplus h = 0, \text{ then } u \leftarrow -1 \text{ and } h' \leftarrow 0 \\ \text{Case 2: If } b_0[1] = 1 \text{ and } b_1[0] \oplus h = 0, \text{ then } u \leftarrow 1 \quad \text{and } h' \leftarrow 0 \\ \text{Case 3: If } b_0[1] = 0 \text{ and } b_1[0] \oplus h = 1, \text{ then } u \leftarrow 1 \quad \text{and } h' \leftarrow 0 \\ \text{Case 4: If } b_0[1] = 1 \text{ and } b_1[0] \oplus h = 1, \text{ then } u \leftarrow -1 \text{ and } h' \leftarrow 1 \end{array} \tag{6}$$

```
Input: Reduced scalar ρ = b_0 + b_1 τ
Output: τ-adic bits (t_{ell-1}, ··· t_0) and flag f
1  f ← assign_flag(b_0[0], b_1[0])
2  (b_0[0], b_1[0]) ← bitflip(b_0[0], b_1[0], f) ;            /* Initial adjustment */
3  i ← 0
4  h ← 0
5  while i < m or |b_0| ≠ 1 or b_1 ≠ 0 do
6  |   (u, h') ← Ψ'(b_0 + b_1 τ) ;                            /* Computed using (6) */
7  |   b_0[1] ← ¬b_0[1] ;     /* Second LSB is set to 1 when Case 1 occurs */
8  |   (b_0, b_1) ← (b_0/2 − b_1 − h, b_0/2)
9  |   t_i ← (−1)^i · u
10 |   h ← h'
11 |_  i ← i + 1
12 t_i ← (−1)^i · b_0
```

Algorithm 4. SPA-resistant generation of a zero-free τ-adic representation

The same technique is also applied to protect the subtraction $d_0 - u$ in the scalar reduction in Algorithm 2.

Protection Against Timing Attack. The terminal condition of the while-loop in Algorithm 3 is dependent on the input scalar. Thus by observing the timing of the computation, an attacker is able to know the higher order bits of a short τ-adic representation. This allows the attacker to narrow down the search domain. We observe that we can continue the generation of zero-free τ-adic bits even when the terminal condition in Algorithm 3 is reached. In this case, the redundant part of the τ-adic representation is equivalent to the value of b_0 when the terminal condition was reached for the first time; hence the result of the point multiplication remains correct. For example, starting from $(b_0, b_1) = (1, 0)$, the algorithm generates an intermediate zero-free representation $-\tau - 1$ and again reaches the terminal condition $(b_0, b_1) = (-1, 0)$. The redundant representation $-\tau^2 - \tau - 1$ is equivalent to 1. If we continue, then the next terminal condition is again reached after generating another two bits. In this paper we generate zero-free τ-adic representations that have lengths always larger than or equal to m of the field $\mathbb{F}_{2^m}$. To implement this feature, we added the terminal condition $i < m$ to the while-loop.

In Algorithm 4, we describe an algorithm for generating zero-free representations that applies the proposed computational optimizations and countermeasures against SPA and timing attacks. The while-loops of both Algorithms 3 and 4 require b_0 to be an odd integer. When the input ρ has an even b_0, then an adjustment is made by adding one to b_0 and adding (subtracting) one to (from) b_1 when b_1 is even (odd). This adjustment is recorded in a flag f in the following way: if b_0 is odd, then $f = 0$; otherwise $f = 1$ or $f = 2$ depending on whether b_1 is even or odd, respectively. In the end of a point multiplication, this flag is checked and $(\tau + 1)P$ or $(-\tau + 1)P$ is subtracted from the point multiplication result if $f = 1$ or $f = 2$, respectively. This compensates the initial addition of $(\tau + 1)$ or $(-\tau + 1)$ to the reduced scalar ρ described in line 2 of Algorithm 4.

```
Input: An integer k, the base point P = (x, y), a random element r ∈ F_{2^m}
Output: The result point Q = kP
1  (t, f) ← Convert(k) ;                                    /* Alg. 2 and 4 */
2  P_{+1} ← φ(P) + P
3  P_{−1} ← φ(P) − P
4  if ℓ is odd then Q = (X, Y) ← t_{ℓ−1}P; i ← ℓ − 3
5  else Q = (X, Y) ← t_{ℓ−1}P_{t_{ℓ−2}t_{ℓ−1}}; i ← ℓ − 4
6  Q = (X, Y, Z) ← (Xr, Yr^2, r)
7  while i ≥ 0 do
8  |   Q ← φ^2(Q)
9  |   Q ← Q + t_{i+1}P_{t_i t_{i+1}}
10 |_  i ← i − 2
11 if f = 1 then Q ← Q + P_{−1}
12 else if f = 2 then Q ← Q − P_{+1}
13 Q = (X, Y) ← (X/Z, Y/Z^2)
14 return Q
```

Algorithm 5. Zero-free point multiplication with side-channel countermeasures

4 Point Multiplication

We base the point multiplication algorithm on the use of the zero-free representation discussed in Sect. 3. We give our modification of the point multiplication algorithm of [32,41] with window size $w = 2$ in Algorithm 5. The algorithm includes countermeasures against SPA, DPA, and timing attacks as well as inherent resistance against safe-error fault attacks. Implementation details of each operation used by Algorithm 5 are given in Appendix A. Below, we give a high-level description.

Line 1 computes the zero-free representation t given an integer k using Algorithms 2 and 4. It outputs a zero-free expansion of length ℓ with $t_i \in \{-1, +1\}$ represented as an ℓ-bit vector and a flag f. Lines 2 and 3 perform the precomputations by computing $P_{+1} = \phi(P) + P$ and $P_{-1} = \phi(P) - P$. Lines 4 and 5 initialize the accumulator point Q depending on the length of the zero-free expansion. If the length is odd, then Q is set to $\pm P$ depending on the msb $t_{\ell-1}$. If the length is even, then Q is initialized with $\pm\phi(P) \pm P$ by using the precomputed points depending on the values of the two msb's $t_{\ell-1}$ and $t_{\ell-2}$. Line 6 randomizes Q by using a random element $r \in \mathbb{F}_{2^m}$ as suggested by Coron [9]. This randomization offers protection against DPA and attacks that calculate hypotheses about the values of Q based on its known initial value (e.g., the doubling attack [12]).

Lines 7 to 10 iterate the main loop of the algorithm by observing two bits of the zero-free expansion on each iteration. Each iteration begins in line 8 by computing two Frobenius endomorphisms. Line 9 either adds or subtracts $P_{+1} = (x_{+1}, y_{+1})$ or $P_{-1} = (x_{-1}, y_{-1})$ to or from Q depending on the values of t_i and t_{i+1} processed by the iteration. It is implemented by using the

equations from [2] which compute a point addition in mixed affine and López-Dahab [27] coordinates. Point addition and subtraction are carried out with the exactly same pattern of operations (see Appendix A). Lines 11 and 12 correct the adjustments that ensure that b_0 is odd before starting the generation of the zero-free representation (see Sect. 3.2). Line 13 retrieves the affine point of the result point Q.

The pattern of operations in Algorithm 5 is almost constant. The side-channel properties of the conversion (line 1) were discussed in Sect. 3. The precomputation (lines 2 and 3) is fixed and operates only on the base point, which is typically public. The initialization of Q (lines 4 and 5) can be carried out with a constant pattern of operations with the help of dummy operations. The randomization of Q protects from differential power analysis (DPA) and comparative side-channel attacks (e.g., the doubling attack [12]). The main loop operates with a fixed pattern of operations on a randomized Q offering protecting against SPA and DPA. Lines 11 and 12 depend on t (and, thus, k) but they leak at most one bit to an adversary who can determine whether they were computed or not. This leakage can be prevented with a dummy operation. Although the algorithm includes dummy operations, it offers good protection also against safe-error fault attacks. The reason is that the main loop does not involve any dummy operations and, hence, even an attacker, who is able to distinguish dummy operations, learns only few bits of information (at most, the lsb and the msb and whether the length is odd or even). Hence, fault attacks that aim to reveal secret information by distinguishing dummy operations are not a viable attack strategy.

5 Architecture

In this section, we describe the hardware architecture (Fig. 1) of our ECC coprocessor for 16-bit microcontrollers such as TI MSP430F241x or MSP430F261x [40]. Such families of low-power microcontrollers have at least 4 KB of RAM and can run at 16 MHz clock. We connect our coprocessor to the microcontroller using a memory-mapped interface [35] following the drop-in concept from [42] where the coprocessor is placed on the bus between the microcontroller and the RAM and memory access is controlled with multiplexers. The coprocessor consists of the following components: an arithmetic and logic unit (ALU), an address generation unit, a shared memory and a control unit composed of hierarchical finite state machines (FSMs).

The Arithmetic and Logic Unit (ECC-ALU) has a 16-bit data path and is used for both integer and binary field computations. The ECC-ALU is interfaced with the memory block using an input register pair (R_1, R_2) and an output multiplexer. The central part of the ECC-ALU consists of a 16-bit integer adder/subtracter circuit, a 16-bit binary multiplier and two binary adders. A small *Reduction-ROM* contains several constants that are used during modular reductions and multiplications by constants. The accumulator register pair (CU, CL) stores the intermediate or final results of any arithmetic operation.

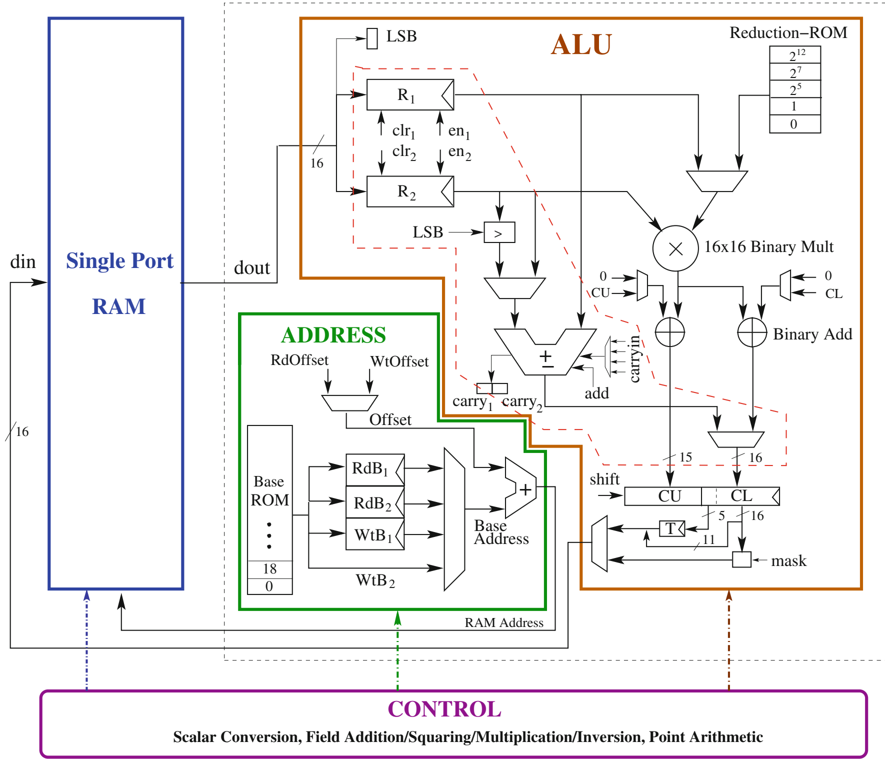

Fig. 1. Hardware architecture of the ECC coprocessor

Finally, the output multiplexer is used to store the contents of the registers CL, T and a masked version of CL in the memory block, which sets the msb's of the most significant word of an alement to zero.

The Memory Block is a single-port RAM which is shared by the ECC coprocessor and the 16-bit microcontroller. Each 283-bit element of $\mathbb{F}_{2^{283}}$ requires 18 16-bit words totaling 288 bits. The coprocessor requires storage for 14 elements of $\mathbb{F}_{2^{283}}$ (see Appendix A), which gives 4032 bits of RAM (252 16-bit words). Some of these variables are reused for different purposes during the conversion.

The Address Unit generates address signals for the memory block. A small *Base-ROM* is used to keep the base addresses for storing different field elements in the memory. During any integer operation or binary field operation, the two address registers RdB_1 and RdB_2 in the address unit are loaded with the base addresses of the input operands. Similarly the base addresses for writing intermediate or final results in the memory block are provided in the register WtB_1

and in the output from the *Base-ROM* (WtB_2). The adder circuit of the address block is an 8-bit adder which computes the physical address from a read/write offset value and a base address.

The Control Unit consists of a set of hierarchical FSMs that generate control signals for the blocks described above. The FSMs are described below.

(1) Scalar Conversion uses the part of the ECC-ALU shown by the red dashed polygon in Fig. 1. The computations controlled by this FSM are mainly integer additions, subtractions and shifts. During any addition or subtraction, the words of the operands are first loaded in the register pair (R_1, R_2). The result-word is computed using the integer adder/subtracter circuit and stored in the accumulator register CL. During a right-shift, R_2 is loaded with the operand-word and R_1 is cleared. Then the lsb of the next higher word of the operand is stored in the one-bit register *LSB*. Now the integer adder is used to add the shifted value $\{LSB, R_2/2\}$ with R_1 to get the shifted word. One scalar conversion requires around 78,000 cycles.

(2) Binary Field Primitives use the registers and the portion of the ECC-ALU outside the red-dashed polygon in Fig. 1.

- Field addition sequentially loads two words of the operands in R_2, then multiplies the words by 1 (from the *Reduction-ROM*) and finally calculates the result-word in CL after accumulation. One field addition requires 60 cycles.
- Field multiplication uses word-serial comb method [13]. It loads the words of the operands in R_1 and R_2, then multiplies the words and finally accumulates. After the completion of the comb multiplication, a modular reduction is performed requiring mainly left-shifts and additions. The left-shifts are performed by multiplying the words with the values from the *Reduction-ROM*. One field multiplication requires 829 cycles.
- Field squaring computes the square of an element of $\mathbb{F}_{2^{283}}$ in linear time by squaring its words. The FSM first loads a word in both R_1 and R_2 and then squares the word by using the binary multiplier. After squaring the words, the FSM performs a modular reduction. The modular reduction is shared with the field multiplication FSM. One field squaring requires 200 cycles.
- Field inversion uses the Itoh-Tsujii algorithm [17] and performs field multiplications and squarings following an addition chain $(1, 2, 4, 8, 16, 17, 34, 35, 70, 140, 141, 282)$ for $\mathbb{F}_{2^{283}}$. One inversion requires 65,241 cycles.

(3) Point Operations and Point Multiplication are implemented by combining an FSM with a hardwired program ROM. The program ROM includes subprograms for all operations of Algorithm 5 and the address of the ROM is controlled by the FSM in order to execute Algorithm 5 (see Appendix A for details).

Algorithm 5 is executed so that the microcontroller initializes the addresses reserved for the accumulator point Q with the base point (x, y) and the random element r by writing $(X, Y, Z) \leftarrow (x, y, r)$. The scalar k is written into the

RAM before the microcontroller issues a start point multiplication command. When this command is received, the reduction part of the conversion is executed followed by the computation of the msb(s) of the zero-free expansion. After this, the precomputations are performed by using (x, y) and the results are stored into the RAM. The initialization of Q is performed by writing either P_{+1} or P_{-1} in (X, Y) if the length of the expansion is even; otherwise, a dummy write is performed. Similarly, the sign of Q is changed if $t_{\ell-1} = -1$ and a dummy operation is computed otherwise. The main loop first executes two Frobenius endomorphisms and, then, issues an instruction that computes the next two bits of the zero-free expansion. By using these bits, either a point addition or a point subtraction is computed with P_{+1} or P_{-1}. One iteration of the main loop takes 9537 clock cycles. In the end, the affine coordinates of the result point are retrieved and they become available for the microcontroller in the addresses for the X and Y coordinates of Q.

6 Results and Comparisons

We described the architecture of Sect. 5 by using mixed Verilog and VHDL and simulated it with ModelSim SE 6.6d. We synthesized the code with Synopsys Design Compiler D-2010.03-SP4 using the regular compile for UMC 130 nm CMOS with voltage of 1.2 V by using Faraday FSC0L low-leakage standard cell libraries. The area given by the synthesis is 4,323 GE including everything in Fig. 1 except the single-port RAM. Computing one point multiplication requires in total 1,566,000 clock cycles including the scalar conversion. The power consumption at 16 MHz is 97.70 μW which gives an energy consumption of approximately 9.56 μJ per point multiplication. Table 1 summarizes our synthesis results together with several other lightweight ECC implementations from the literature.

Among all lightweight ECC processors available in the literature, the processor from [4] is the closest counterpart to our implementation because it is so far the only one that uses Koblitz curves. Even it has many differences with our architecture which make fair comparison difficult. The most obvious difference is that the processor from [4] is designed for a less secure Koblitz curve NIST K-163. Also the architecture of [4] differs from ours in many fundamental ways: they use a finite field over normal basis instead of polynomial basis, they use a bit-serial multiplier that requires all bits of both operands to be present during the entire multiplication instead of a word-serial architecture that we use, they store all variables in registers embedded into the processor architecture instead of an external RAM, and they also do not provide support for scalar conversions or any countermeasures against side-channel attacks. They also provide implementation results on 65 nm CMOS. Our architecture is significantly more scalable for different Koblitz curves because, besides control logic and RAM requirements, other parts remain almost the same, whereas the entire multiplier needs to be changed for [4]. It is also hard to see how scalar conversions or side-channel countermeasures could be integrated into the architecture of [4] without significant increases on both area and latency.

Table 1. Comparison to other lightweight coprocessors for ECC. The top part consists of relevant implementations from the literature. We also provide estimates for other parameter sets in order to ease comparisons to existing works.

Work	Curve	Conv	RAM	Tech. (nm)	Freq. (MHz)	Area (GE)	Latency (cycles)	Latency (ms)	Power (μW[a])
[5], 2006	B-163	n/a	no	130	0.500	9,926	95,159	190.32	<60
[6], 2008	B-163	n/a	yes	220	0.847	12,876	–	95	93
[15], 2008	B-163	n/a	yes	180	0.106	13,250	296,299	2,792	80.85
[24], 2006	B-163	n/a	yes	350	13.560	16,207	376,864	27.90	n/a
[26], 2008	B-163	n/a	yes	130	1.130	12,506	275,816	244.08	32.42
[43], 2011	B-163	n/a	yes	130	0.100	8,958	286,000	2,860	32.34
[42], 2013	B-163	n/a	no	130	1.000	4,114	467,370	467.37	66.1
[34], 2014	P-160	n/a	yes	130	1.000	12,448[b]	139,930	139.93	42.42
[4], 2014	K-163	no	yes[c]	65	13.560	11,571	106,700	7.87	5.7
Our, est	B-163	yes	no	130	16.000	≈3,773	≈485,000	≈30.31	≈6.11
Our, est	K-163	yes	no	130	16.000	≈4,323	≈420,900	≈26.30	≈6.11
Our, est	B-283	yes	no	130	16.000	≈3,773	≈1,934,000	≈120.89	≈6.11
Our, est	K-283	yes	yes[d]	130	16.000	10,204	1,566,000	97.89	>6.11
Our	K-283	yes	no	130	16.000	4,323	1,566,000	97.89	6.11

[a] Normalized to 1 MHz.
[b] Contains everything required for ECDSA including a Keccak module.
[c] All variables are stored in registers inside the processor.
[d] The 256 × 16-bit RAM is estimated to have an area of 5794 GE because the size of a single-port 256 × 8-bit RAM has an area of 2897 GE [42].

Table 1 includes also implementations that use the binary curve B-163 and the prime curve P-160 from [31]. The area of our coprocessor is on the level of the smallest coprocessors available in the literature. Hence, the effect of selecting a 283-bit elliptic curve instead of a less secure curve is negligible in terms of area. The price to pay for higher security comes in the form of memory requirements and computation latency. The amount of memory is not a major issue because our processor shares the memory with the microcontroller which typically has a large memory (e.g. TI MSP430F241x and MSP430F261x have at least 4 KB RAM [40]). Also the computation time is on the same level with other published implementations because our coprocessor is designed to run on the relatively high clock frequency of the microcontroller which is 16 MHz.

In this work our main focus was to investigate feasibility of lightweight implementations of Koblitz curves for applications demanding high security. To enable a somewhat fair comparison with the existing lightweight implementations over $\mathbb{F}_{2^{163}}$, Table 1 provides estimates (see Appendix B) for area and cycles of ECC coprocessors that follow the design decisions presented in this paper and perform point multiplications on curves B-163 or K-163. Our estimates show that our coprocessors for both B-163 and K-163 require more cycles in comparison

to [43] which also uses a 16-bit ALU. The reason behind this is that [43] uses a dual-port RAM, whereas our implementation uses a single-port RAM (as it works as a coprocessor of MSP430). Moreover [43] has a dedicated squarer circuit to minimize cycle requirement for squaring.

Table 1 provides estimates for cycle and area of a modified version of the coprocessor that performs point multiplications using the Montgomery's ladder on the NIST curve B-283. The estimated cycle count is calculated from the cycle counts of the field operations described in Sect. 5. From the estimated value, we see that a point multiplication on B-283 requires nearly 23.5 % more time. However, the coprocessor for B-283 is smaller by around 550 GE as no scalar conversion is needed.

Although application-specific integrated circuits are the primary targets for our coprocessor, it may be useful also for FPGA-based implementations whenever small ECC designs are needed. Hence, we compiled our coprocessor also for Xilinx Spartan-6 XC6SLX4-2TQG144 FPGA by using Xilinx ISE 13.4 Design Suite. After place & route, it requires only 209 slices (634 LUTs and 309 registers) and runs on clock frequencies up to 106.598 MHz.

Our coprocessor significantly improves speed, both classical and side-channel security, memory footprint, and energy consumption compared to leading lightweight software [3,10,16,21,38]. For example, [10] reports a highly optimized Assembly implementation running on a 32-bit Cortex-M0+ processor clocked at 48 MHz that computes a point multiplication on a less secure Koblitz curve K-233 without strong side-channel countermeasures. It computes a point multiplication in 59.18 ms (177.54 ms at 16 MHz) and consumes 34.16 μJ of energy.

7 Conclusions

In this paper we showed that implementing point multiplication on a high security 283-bit Koblitz curve is feasible with extremely low resources making it possible for various lightweight applications. We also showed that Koblitz curves can be used in such applications even when the cryptosystem requires scalar conversions. Beside these contributions, we improved the scalar conversion by applying several optimizations and countermeasures against side-channel attacks. Finally, we designed a very lightweight architecture in only 4.3 kGE that can be used as a coprocessor for commercial 16-bit microcontrollers. Hence, we showed that Koblitz curves are feasible also for lightweight ECC even with on-the-fly scalar conversions and strong countermeasures against side-channel attacks.

Acknowlegments. S. Sinha Roy was supported by the Erasmus Mundus PhD Scholarship and K. Järvinen was funded by FWO Pegasus Marie Curie Fellowship. This work was supported by the Research Council KU Leuven: TENSE (GOA/11/007), by iMinds, by the Flemish Government, FWO G.0550.12N, G.00130.13N and FWO G.0876.14N, and by the Hercules Foundation AKUL/11/19. We thank Bohan Yang for his help with ASIC synthesis and simulations.

A Implementation of Operations Used by Algorithm 5

The operations required by Algorithm 5 are implemented by combining an FSM and a program ROM. The program ROM includes subprograms for all operations of Algorithm 5 and the FSM sets the address of the ROM to the first instruction of the subprogram according the phase of the algorithm and t_{i+1}, t_i.

Table 2 shows the contents of the program ROM. The operations required by Algorithm 5 are in this ROM as follows:

- Line 0 obtains the next bits of the zero-free representation.
- Lines 1–23 perform the precomputation that computes $(x_{+1}, y_{+1}) = \phi(P) + P$ and $(x_{-1}, y_{-1}) = \phi(P) - P$.
- Line 24 computes the negative of Q during the initialization and Line 25 is the corresponding dummy operation.
- Lines 26–28 randomize the projective coordinates of Q by using the random $r \in \mathbb{F}_{2^{283}}$ which is stored in Z.
- Lines 29–34 compute two Frobenius endomorphisms for Q.
- Lines 35–37 set $(x_p, y_p) \leftarrow (x_{+1}, y_{+1}) = \phi(P) + P$ and compute the y-coordinate of its negative to y_m.

Table 2. The program ROM includes instructions for the following operations

0 Convert(k)	**20** $y_{-1} \leftarrow y_{-1} \times T_1$	**40** $y_m \leftarrow x_{-1} + y_{-1}$	**60** $T_2 \leftarrow x_p \times Z$
1 $x_{+1} \leftarrow X^2$	**21** $y_{-1} \leftarrow y_{-1} + x_{-1}$	**41** $x_p \leftarrow x_{+1}$	**61** $T_2 \leftarrow T_2 + X$
2 $y_{+1} \leftarrow Y^2$	**22** $y_{-1} \leftarrow y_{-1} + Y$	**42** $y_m \leftarrow y_{+1}$	**62** $Y \leftarrow Y + Z$
3 $x_{+1} \leftarrow X + x_{+1}$	**23** $y_{-1} \leftarrow y_{-1} + X$	**43** $y_p \leftarrow x_{+1} + y_{+1}$	**63** $Y \leftarrow Y \times T_2$
4 $x_{-1} \leftarrow x_{+1}^{-1}$	**24** $Y \leftarrow X + Y$	**44** $x_p \leftarrow x_{-1}$	**64** $T_1 \leftarrow Z^2$
5 $T_1 \leftarrow Y + y_{+1}$	**25** $T_1 \leftarrow X + Y$	**45** $y_m \leftarrow y_{-1}$	**65** $T_1 \leftarrow T_1 \times y_m$
6 $y_{-1} \leftarrow T_1 \times x_{-1}$	**26** $X \leftarrow X \times Z$	**46** $y_p \leftarrow x_{-1} + y_{-1}$	**66** $Y \leftarrow Y + T_1$
7 $T_1 \leftarrow y_{-1}^2$	**27** $T_1 \leftarrow Z^2$	**47** $T_1 \leftarrow Z^2$	**67** $x_{+1} \leftarrow Z$
8 $T_1 \leftarrow T_1 + y_{-1}$	**28** $Y \leftarrow Y \times T_1$	**48** $T_1 \leftarrow T_1 \times y_p$	**68** $x_{-1} \leftarrow x_{+1}^{-1}$
9 $x_{+1} \leftarrow T_1 + x_{+1}$	**29** $Y \leftarrow Y^2$	**49** $T_1 \leftarrow T_1 + Y$	**69** $X \leftarrow X \times x_{-1}$
10 $T_1 \leftarrow x_{+1} + X$	**30** $Y \leftarrow Y^2$	**50** $T_2 \leftarrow Z \times x_p$	**70** $x_{-1} \leftarrow x_{-1}^2$
11 $y_{+1} \leftarrow T_1 + y_{-1}$	**31** $X \leftarrow X^2$	**51** $T_2 \leftarrow T_2 + X$	**71** $Y \leftarrow Y \times x_{-1}$
12 $y_{+1} \leftarrow y_{+1} + x_{+1}$	**32** $X \leftarrow X^2$	**52** $X \leftarrow T_2^2$	**72** $X \leftarrow x_{+1}$
13 $y_{+1} \leftarrow y_{+1} + Y$	**33** $Z \leftarrow Z^2$	**53** $X \leftarrow X + T_1$	**73** $Y \leftarrow y_{+1}$
14 $x_{-1} \leftarrow x_{-1} \times X$	**34** $Z \leftarrow Z^2$	**54** $T_2 \leftarrow T_2 \times Z$	**74** $X \leftarrow x_{-1}$
15 $y_{-1} \leftarrow y_{-1} + x_{-1}$	**35** $x_p \leftarrow x_{+1}$	**55** $X \leftarrow X \times T_2$	**75** $Y \leftarrow y_{-1}$
16 $T_1 \leftarrow x_{-1}^2$	**36** $y_p \leftarrow y_{+1}$	**56** $Y \leftarrow T_1 \times T_2$	**76** $T_1 \leftarrow x_{+1}$
17 $x_{-1} \leftarrow x_{-1} + T_1$	**37** $y_m \leftarrow x_{+1} + y_{+1}$	**57** $T_1 \leftarrow T_1^2$	**77** $T_2 \leftarrow y_{+1}$
18 $x_{-1} \leftarrow x_{-1} + x_{+1}$	**38** $x_p \leftarrow x_{-1}$	**58** $X \leftarrow X + T_1$	
19 $T_1 \leftarrow x_{-1} + X$	**39** $y_p \leftarrow y_{-1}$	**59** $Z \leftarrow T_2^2$	

– Lines 38–40 set $(x_p, y_p) \leftarrow (x_{-1}, y_{-1}) = \phi(P) - P$ and compute the y-coordinate of its negative to y_m.
– Lines 41–43 compute $(x_p, y_p) \leftarrow -(x_{+1}, y_{+1}) = -\phi(P) - P$ and set the y-coordinate of its negative to y_m.
– Lines 44–46 compute $(x_p, y_p) \leftarrow -(x_{-1}, y_{-1}) = -\phi(P) + P$ and set the y-coordinate of its negative to y_m.
– Lines 47–66 compute the point addition $(X, Y, Z) \leftarrow (X, Y, Z) + (x_p, y_p)$ in López-Dahab coordinates using the equations from [2].
– Lines 67–71 recover the affine coordinates of Q by computing $(X, Y) \leftarrow (X/Z, Y/Z^2)$.
– Lines 72–73 and lines 74–75 initialize Q with (x_{+1}, y_{+1}) and (x_{-1}, y_{-1}), respectively, and lines 76–77 perform a dummy operation for these operations.

Point addition and point subtraction are computed with exactly the same sequence of operations. This is achieved by introducing an initialization which sets the values of three internal variables x_p, y_p, and y_m according to Table 3 (these are in lines 35–46 in Table 2). This always requires two copy instructions followed by an addition. After this initialization, both point addition and point subtraction are computed with a common sequence of operations which adds the point (x_p, y_p) to Q. The element x_m is the y-coordinate of the negative of (x_p, y_p) and it is also used during the point addition.

Table 3. Initialization of point addition and point subtraction

t_{i+1}, t_i	1st	2nd	3rd
$+1, +1$	$x_p \leftarrow x_{+1}$	$y_p \leftarrow y_{+1}$	$y_m \leftarrow x_{+1} + y_{+1}$
$+1, -1$	$x_p \leftarrow x_{-1}$	$y_p \leftarrow y_{-1}$	$y_m \leftarrow x_{-1} + y_{-1}$
$-1, +1$	$x_p \leftarrow x_{-1}$	$y_m \leftarrow y_{-1}$	$y_p \leftarrow x_{-1} + y_{-1}$
$-1, -1$	$x_p \leftarrow x_{+1}$	$y_m \leftarrow y_{+1}$	$y_p \leftarrow x_{+1} + y_{+1}$

B Estimates for B-163 and K-163

Our estimated cycle count for scalar multiplication over $\mathbb{F}_{2^{163}}$ is based on the following facts:

1. A field element in $\mathbb{F}_{2^{163}}$ requires 11 16-bit words, and hence, is smaller by a factor of 0.61 than a field element in $\mathbb{F}_{2^{283}}$. Since field addition and squaring have linear complexity, we estimate that the cycle counts for these operations scale down by a factor of around 0.61 and become 37 and 122 respectively. In a similarly way we estimate that field multiplication (which has quadratic complexity) scales down to 309 cycles. A field inversion operation following an addition chain $(1, 2, 4, 5, 10, 20, 40, 81, 162)$ requires nearly 22,700 cycles.

2. The for-loop in the scalar reduction operation (Algorithm 2) executes 163 times in $\mathbb{F}_{2^{163}}$ and performs linear operations such as additions/subtractions and shifting. Moreover the length of τ-adic representation of a scalar reduces to 163 (thus reducing by a factor of 0.57 in comparison to $\mathbb{F}_{2^{283}}$). So, we estimate that the cycle count for scalar conversion scales down by a factor of 0.57×0.61 and requires nearly 27,000 cycles.
3. One Frobenius-and-add operation over $\mathbb{F}_{2^{283}}$ in Algorithm 5 spends total 9,537 cycles among which 6,632 cycles are spent in eight quadratic-time field multiplications, and the rest 2,905 cycles are spent in linear-time operations. After scaling down, the cycle count for one Frobenius-and-add operation over $\mathbb{F}_{2^{163}}$ can be estimated to be around 4,250. The point multiplication loop iterates nearly 82 times for a τ-adic representation of length 164. Hence the number of cycles spent in this loop can be estimated to be around 348,500.
4. The precomputation and the final conversion steps are mainly dominated by the cost of field inversions. Hence the cycle counts can be estimated to be around 45,400.

As per the above estimates we see that a point multiplication using K-163 requires nearly 420,900 cycles. Similarly, we estimate that Montgomery's ladder for B-163 requires nearly 485,000 cycles.

References

1. Adikari, J., Dimitrov, V., Järvinen, K.: A fast hardware architecture for integer to τNAF conversion for Koblitz curves. IEEE Trans. Comput. **61**(5), 732–737 (2012)
2. Al-Daoud, E., Mahmod, R., Rushdan, M., Kilicman, A.: A new addition formula for elliptic curves over $GF(2^n)$. IEEE Trans. Comput. **51**(8), 972–975 (2002)
3. Aranha, D.F., Dahab, R., López, J., Oliveira, L.B.: Efficient implementation of elliptic curve cryptography in wireless sensors. Adv. Math. Commun. **4**(2), 169–187 (2010)
4. Azarderakhsh, R., Järvinen, K.U., Mozaffari-Kermani, M.: Efficient algorithm and architecture for elliptic curve cryptography for extremely constrained secure applications. IEEE Trans. Circ. Syst. I–Regul. Pap. **61**(4), 1144–1155 (2014)
5. Batina, L., Mentens, N., Sakiyama, K., Preneel, B., Verbauwhede, I.: Low-cost elliptic curve cryptography for wireless sensor networks. In: Buttyán, L., Gligor, V.D., Westhoff, D. (eds.) ESAS 2006. LNCS, vol. 4357, pp. 6–17. Springer, Heidelberg (2006)
6. Bock, H., Braun, M., Dichtl, M., Hess, E., Heyszl, J., Kargl, W., Koroschetz, H., Meyer, B., Seuschek, H.: A milestone towards RFID products offering asymmetric authentication based on elliptic curve cryptography. In: Proceedings of the 4th Workshop on RFID Security – RFIDSec 2008 (2008)
7. Brumley, B.B., Järvinen, K.U.: Koblitz curves and integer equivalents of frobenius expansions. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 126–137. Springer, Heidelberg (2007)
8. Brumley, B.B., Järvinen, K.U.: Conversion algorithms and implementations for Koblitz curve cryptography. IEEE Trans. Comput. **59**(1), 81–92 (2010)
9. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)

10. De Clercq, R., Uhsadel, L., Van Herrewege, A., Verbauwhede, I.: Ultra low-power implementation of ECC on the ARM Cortex-M0+. In: Design Automation Conference – DAC 2014, pp. 1–6. ACM (2014)
11. Fan, J., Verbauwhede, I.: An updated survey on secure ECC implementations: attacks, countermeasures and cost. In: Naccache, D. (ed.) Cryphtography and Security: From Theory to Applications. LNCS, vol. 6805, pp. 265–282. Springer, Heidelberg (2012)
12. Fouque, P.-A., Valette, F.: The doubling attack – *why upwards is better than downwards*. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 269–280. Springer, Heidelberg (2003)
13. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus (2003)
14. Hasan, M.A.: Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems. IEEE Trans. Comput. **50**(10), 1071–1083 (2001)
15. Hein, D., Wolkerstorfer, J., Felber, N.: ECC is ready for RFID – a proof in silicon. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 401–413. Springer, Heidelberg (2009)
16. Hinterwälder, G., Moradi, A., Hutter, M., Schwabe, P., Paar, C.: Full-size high-security ECC implementation on MSP430 microcontrollers. In: Aranha, D.F., Menezes, A. (eds.) LATINCRYPT 2014. LNCS, vol. 8895, pp. 31–47. Springer, Heidelberg (2015)
17. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Inf. Comput. **78**(3), 171–177 (1988)
18. Järvinen, K.: Optimized FPGA-based elliptic curve cryptography processor for high-speed applications. Integr. VLSI J. **44**(4), 270–279 (2011)
19. Järvinen, K., Forsten, J., Skyttä, J.: Efficient circuitry for computing τ-adic non-adjacent form. In: Proceedings of the 13th IEEE International Conference on Electronics, Circuits and Systems – ICECS 2006, pp. 232–235. IEEE (2006)
20. Järvinen, K., Verbauwhede, I.: How to use Koblitz curves on small devices? In: Joye, M., Moradi, A. (eds.) CARDIS 2014. LNCS, vol. 8968, pp. 154–170. Springer, Heidelberg (2015)
21. Kargl, A., Pyka, S., Seuschek, H.: Fast arithmetic on ATmega128 for elliptic curve cryptography. Cryptology ePrint Archive, Report 2008/442 (2008)
22. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987)
23. Koblitz, N.: CM-curves with good cryptographic properties. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 279–287. Springer, Heidelberg (1992)
24. Kumar, S., Paar, C.: Are standards compliant elliptic curve cryptosystems feasible on RFID? In: Handouts of the Workshop on RFID Security – RFIDSec 2006 (2006)
25. Lange, T.: Koblitz curve cryptosystems. Finite Fields Appl. **11**, 200–229 (2005)
26. Lee, Y.K., Sakiyama, K., Batina, L., Verbauwhede, I.: Elliptic-curve-based security processor for RFID. IEEE Trans. Comput. **57**(11), 1514–1527 (2008)
27. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in $GF(2^n)$. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 201–212. Springer, Heidelberg (1999)
28. Meier, W., Staffelbach, O.: Efficient multiplication on certain nonsupersingular elliptic curves. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 333–344. Springer, Heidelberg (1993)
29. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)

30. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**, 243–264 (1987)
31. National Institute of Standards and Technology (NIST): Digital signature standard (DSS). Federal Information Processing Standard, FIPS PUB 186–4, July 2013
32. Okeya, K., Takagi, T., Vuillaume, C.: Efficient representations on Koblitz curves with resistance to side channel attacks. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 218–229. Springer, Heidelberg (2005)
33. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Lambda coordinates for binary elliptic curves. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 311–330. Springer, Heidelberg (2013)
34. Pessl, P., Hutter, M.: Curved tags — a low-resource ECDSA implementation tailored for RFID. In: Sadeghi, A.-R., Saxena, N. (eds.) RFIDSec 2014. LNCS, vol. 8651, pp. 156–172. Springer, Heidelberg (2014)
35. Schaumont, P.R.: A Practical Introduction to Hardware/Software Codesign, 2nd edn. Springer, US (2013)
36. Sinha Roy, S., Fan, J., Verbauwhede, I.: Accelerating scalar conversion for Koblitz curve cryptoprocessors on hardware platforms. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **23**(5), 810–818 (2015)
37. Solinas, J.A.: Efficient arithmetic on Koblitz curves. Des. Codes Crypt. **19**(2–3), 195–249 (2000)
38. Szczechowiak, P., Oliveira, L.B., Scott, M., Collier, M., Dahab, R.: NanoECC: testing the limits of elliptic curve cryptography in sensor networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008)
39. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. J. Cryptographic Eng. **1**(3), 187–199 (2011)
40. Texas Instruments: MSP430F261x and MSP430F241x, Jun 2007, Rev Nov 2012. http://www.ti.com/lit/ds/symlink/msp430f2618.pdf. Accessed 4 June 2015
41. Vuillaume, C., Okeya, K., Takagi, T.: Defeating simple power analysis on Koblitz curves. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E89–A**(5), 1362–1369 (2006)
42. Wenger, E.: Hardware architectures for MSP430-based wireless sensor nodes performing elliptic curve cryptography. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 290–306. Springer, Heidelberg (2013)
43. Wenger, E., Hutter, M.: A hardware processor supporting elliptic curve cryptography for less than 9 kGEs. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 182–198. Springer, Heidelberg (2011)