Kimmo U. Järvinen and Jorma O. Skyttä, High-Speed Elliptic Curve Cryptography Accelerator for Koblitz Curves, in Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2008, Stanford, California, USA, Apr. 14-15, 2008, in press, 10 pages.

High-Speed Elliptic Curve Cryptography Accelerator for Koblitz Curves

Kimmo U. Järvinen and Jorma O. Skyttä

Helsinki University of Technology
Department of Signal Processing and Acoustics
Otakaari 5A, FIN-02150 Espoo, Finland
E-mail: {kimmo.jarvinen, jorma.skytta}@tkk.fi

Abstract

We present an FPGA-based accelerator for elliptic curve cryptography on a Koblitz curve targeting for applications requiring very high speed. The accelerator supports fast computation of point multiplication by using window methods as well as multiple point multiplications with joint sparse form representations. Optimized operation-specific processing units are used in order to improve performance. Throughput is increased by pipelining operations. The accelerator was implemented in an Altera Stratix II FPGA and it computes point multiplication on average in 16.36 μs and achieves a maximum of 161,290 operations per second. A 3-term multiple point multiplication requires 35.06 μs with a maximum of 60,603 operations in second.

1 Introduction

Elliptic curve cryptography has been an active area of research since 1985 when Koblitz [18] and Miller [23] independently suggested using elliptic curves for public-key cryptography. Because elliptic curve cryptography offers the same level of security than, for example, RSA with considerably shorter keys, it has replaced traditional public-key cryptosystems, especially, in environments where short keys are important. Public-key cryptosystems are computationally demanding and, hence, the fact that elliptic curve cryptography has been shown to be faster than traditional public-key cryptosystems (see [11], for example) is of great importance. Speed can be further increased by using a special class of elliptic curves referred to as Koblitz curves [19]. These curves are included in many standards; see [6, 24], for example.

We discuss FPGA-based implementation of elliptic curve cryptography which has attained considerable interest in both cryptography and FPGA communities. The reason for this interest is that public-key cryptosystems are computationally demanding which often yields a need for hardware acceleration. FPGAs are feasible alternatives for cryptographic implementations because of the combination of fast performance and flexibility [27]. Furthermore, FPGAs allow optimizing designs for certain fixed parameters because parameter flexibility can be achieved through reprogrammability. This gives a major advantage over ASIC-based implementations where such optimizations are usually out of reach. A comprehensive survey of implementing elliptic curve cryptography is given in [22].

The primary application for our accelerator is the *packet level authentication* (PLA) communication scheme [5] where IP packets are signed and verified by using cryptographic signatures. Because verification is performed from node to node [5], computational requirements are enormous. Hence, we use Koblitz curves in order to maximize performance. Previously FPGA-based implementations using Koblitz curves have been presented in [9, 15–17, 21, 25] and they are reviewed in Sec. 2.

Traditionally papers discussing elliptic curve cryptography in FPGAs have concentrated solely on minimizing the computation time of a single elliptic curve point multiplication, the fundamental operation of elliptic curve cryptosystems. The growth in available logic resources in FPGAs has enabled using large amounts of parallelism. However, the problem is that elliptic curve point multiplication is hard to parallelize because of its sequential nature. Although parallelism can be used in lower hierarchy levels, using large amounts of parallelism usually leads to poor latency-area products [17]. In our recent work, we showed how relaxing the computation time constraints slightly results in major enhancements in throughput (operations per second), because more parallel processing units fit into an FPGA [15].

Our new architecture combines several architectures from our previous works, [14–17], in a novel way and provides both short computation times and high throughputs. Hence, the main contribution of our paper is in the way

This work was supported in part by the project "Packet Level Authentication" funded by TEKES.

how dedicated processing units are used on the top level to provide fast performance, not in the processing units themselves because they have been presented in our earlier publications. The key idea is to use specialized processing units for different parts of algorithms and increase throughput by pipelining their computation. The architecture supports computations of sums of up to three point multiplications which are commonly required in many elliptic curve cryptosystems and utilizes window algorithms with precomputations in order to increase speed. High performance and area efficiency are achieved for these operations by efficiently exploiting their common structure. To the best of our knowledge, our accelerator outperforms all previously published implementations.

The remainder of the paper is organized as follows. Related work is reviewed in Sec. 2. Preliminaries of elliptic curve cryptography and algorithms implemented by the accelerator are introduced in Sec. 3. Sec. 4 describes the accelerator architecture in detail and an analysis and optimizations are presented in Sec 5. Results are presented and compared to existing work in Sec. 6. We end with conclusions and discussion on future work in Sec. 7.

2 Related Work

The first FPGA-based implementation using Koblitz curves was presented in 2000 by Okada *et al.* in [25], where one point multiplication was shown to require 45.6 ms on a standardized curve NIST K-163 [24] with an Altera Flex 10K FPGA. They concluded that Koblitz curves are approximately twice as fast as general curves. In 2004, Lutz and Hasan [21] presented an implementation which computes point multiplication in 75 μs on NIST K-163 in a Xilinx Virtex-E FPGA. Neither of the two designs includes a circuitry for conversions that are mandatory for Koblitz curves (see Sec. 3.3). In 2006, Dimitrov *et al.* [9] proposed a double-base expansion which can be used for increasing the speed of Koblitz curve computations and presented FPGA implementations for both elliptic curve point multiplication and conversion. Elliptic curve point multiplication was shown to require 35.75 μs on NIST K-163 with a Xilinx Virtex-II.

Our recent work for Koblitz curves in FPGAs consists of [14–17]. An efficient circuitry for computing the conversions was presented in [14]. It was shown in [15] that up to 166,000 signature verifications in the PLA can be computed using a single Stratix II FPGA with parallel processing. More general parallelization studies were presented in [17] and they resulted in an implementation that computes point multiplication in only 25.81 μs. Recently, we showed that even shorter computation time of only 4.91 μs can be achievable on NIST K-163 with interleaved operations [16].

3 Preliminaries

3.1 Finite Fields

Elliptic curves defined over finite fields $\mathbb{F}_q$ are used in cryptography and only curves over *binary fields*, where $q = 2^m$, with *polynomial basis* are considered in this paper. Polynomial bases are commonly used in elliptic curve cryptosystems because they provide fast performance on both software and hardware. Another commonly used basis, normal basis, provides very efficient squaring but multiplication is more complicated. We base our selection on our recent study which favored polynomial basis [16].

Elements of $\mathbb{F}_{2^m}$ with polynomial basis are represented as binary polynomials with degrees less than m as $a(x) = \sum_{i=0}^{m-1} a_i x^i$. Arithmetic operations in $\mathbb{F}_{2^m}$ are computed modulo an irreducible polynomial[1] with a degree m. Because sparse polynomials offer considerable computational advantages, trinomials (three nonzero terms) or pentanomials (five nonzero terms) are used in practice. The curve, NIST K-163, considered in this paper is defined over $\mathbb{F}_{2^{163}}$ with the pentanomial $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$ [24].

Addition, $a(x) + b(x)$, in $\mathbb{F}_{2^m}$ is a bitwise exclusive-or (XOR). *Multiplication*, $a(x)b(x)$, is more involved and it consists of two steps: ordinary multiplication of polynomials and reduction modulo $p(x)$. If both multiplicands are the same, the operation is called squaring, $a^2(x)$. *Squaring* is cheaper than multiplication because the multiplication of polynomials is performed simply by adding zeros to the bit vector. Reduction modulo $p(x)$ can be performed with a small number of XORs if $p(x)$ is sparse and fixed, i.e. the same $p(x)$ is always used, which is the case in this paper. *Inversion*, $a^{-1}(x)$, is an operation which finds $b(x)$ such that $a(x)b(x) = 1$ when $a(x)$ is given. Inversion is the most complex operation and it can be computed either with the Extended Euclidean Algorithm or Fermat's Little Theorem that gives $a^{-1}(x) = a^{2^m-2}(x)$.

Multiplication has the most crucial effect on performance of an elliptic curve cryptosystem. A *digit-serial multiplier* computes D bits of the output in one cycle resulting in a total latency of $\lceil m/D \rceil$ cycles. We use hardware modifications of the multiplier described in [12]. Instead of using precomputed look-up tables as in [12], our multiplier computes everything on-the-fly similarly as in [16, 21].

3.2 Elliptic Curve Point Multiplication

Let E be an *elliptic curve* defined over a finite field $\mathbb{F}_q$. Points on E form an additive Abelian group, $E(\mathbb{F}_q)$, together with a point called the *point at infinity*, $\mathcal{O}$, which

[1] A polynomial, $f(x) \in \mathbb{F}[x]$, with a positive degree is irreducible over $\mathbb{F}$ if it cannot be presented as a product of two polynomials in $\mathbb{F}[x]$ with positive degrees.

acts as a zero element. The group operation is called point addition.

Elliptic curve point multiplication is defined by

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

where k is a positive integer and $P, Q \in E(\mathbb{F}_q)$. The security of elliptic curve cryptosystems is based on the assumption that it is computationally infeasible to find k if P and Q are known if E is chosen carefully.

Two basic operations are used in computing (1): *point addition* and *point doubling*. Point addition refers to the operation $P_3 = P_1 + P_2$ where $P_i \in E(\mathbb{F}_q)$ so that $P_1 \neq P_2$. Point doubling is the operation $P_3 = P_1 + P_1 = 2P_1$. Point subtraction is simply $P_3 = P_1 + (-P_2)$ where $-P_2 = (x_2, x_2 + y_2)$ for $P_2 = (x_2, y_2)$ in $E(\mathbb{F}_{2^m})$.

Arguably, the simplest algorithm for computing (1) is the *double-and-add algorithm* (binary algorithm). The algorithm uses k given with binary expansion as $k = \sum_{i=0}^{\ell-1} k_i 2^i$ where $k_i \in \{0, 1\}$ and $\ell \approx \lceil \log_2 q \rceil$. The algorithm operates on the bits of k sequentially starting either from the lsb or the msb of k. Each bit results in a point doubling whereas a point addition is needed only if $k_i = 1$. The number of nonzeros in a representation of k is called *Hamming weight* and denoted by $H(k)$. Because $H(k) \approx \ell/2$, the double-and-add algorithm requires ℓ point doublings and, on average, $\ell/2$ point additions.

If points are represented traditionally with two coordinates as (x, y), both point addition and doubling require an inversion in $\mathbb{F}_{2^m}$. This coordinate system is referred to as *affine coordinates*, or $\mathcal{A}$ for short. Inversions are expensive and, thus, projective coordinates of the form (X, Y, Z) are commonly used in practical implementations, because then point addition and point doubling are computed with only multiplications, squarings, and additions. However, because the result point is needed in $\mathcal{A}$, the point must be converted in the end. In this paper, we use the so-called *López-Dahab coordinates*, or $\mathcal{LD}$ for short, where a point (X, Y, Z) represents the affine point [20]

$$(x, y) = (X/Z, Y/Z^2) \ . \tag{2}$$

Very efficient point addition formulae exist when P_1 is in $\mathcal{LD}$ and P_2 is in $\mathcal{A}$ [1].The formulae for $(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (x_2, y_2)$ are as follows [1]:

$$\begin{gathered} A = Y_1 + y_2 Z_1^2; \quad B = X_1 + x_2 Z_1 \\ C = BZ_1; \quad Z_3 = C^2; \quad D = x_2 Z_3 \\ X_3 = A^2 + C(A + B^2 + aC) \\ Y_3 = (D + X_3)(AC + Z_3) + (y_2 + x_2) Z_3^2 \end{gathered} \tag{3}$$

where a is a curve paramater; see Sec. 3.3.

3.3 Koblitz Curves

Koblitz curves [19] are a family of elliptic curves defined over $\mathbb{F}_{2^m}$ by the following equation:

$$E_K : y^2 + xy = x^3 + ax^2 + 1 \tag{4}$$

where $a \in \{0, 1\}$. Koblitz curves are appealing because they offer considerable computational advantages over general curves. These advantages are based on the fact that an algorithm, similar to double-and-add, can be devised so that point doublings are replaced by *Frobenius endomorphisms*. The Frobenius endomorphism, ϕ, for a point $P = (x, y)$ is a map such that

$$\phi : (x, y) \mapsto (x^2, y^2) \quad \text{and} \quad \mathcal{O} \mapsto \mathcal{O} \ . \tag{5}$$

Obviously, Frobenius endomorphism is very cheap: only two or three squarings depending on the coordinate system.

Replacing point doublings with Frobenius endomorphisms is not straightforward, but requires manipulations on k. It stands for all points in $E_K(\mathbb{F}_{2^m})$ that $\mu\phi(P) - \phi^2(P) = 2P$ where $\mu = (-1)^{1-a}$. Thus, ϕ can be seen as a complex number, τ, satisfying $\mu\tau - \tau^2 = 2$ which gives $\tau = (\mu + \sqrt{-7})/2$. Moving from a bit to another in a representation of k corresponds to an application of ϕ if k is given in a *τ-adic representation* as $k = \sum_{i=0}^{\ell-1} k_i \tau^i$. Hence in order to utilize fast Frobenius endomorphisms, k must be converted into a τ-adic representation. [19]

A thorough study of the conversion and efficient conversion algorithms were presented by Solinas in [26]. The basic algorithm returns the so-called *τ-adic non-adjacent form* (τNAF) where k is represented with the signed-binary format, i.e. $k_i \in \{0, \pm 1\}$. Henceforth, we denote $\bar{1} = -1$. The average length of τNAF is the same as the binary length of k, i.e. ℓ. τNAF has $H(k) \approx \ell/3$ and one of two adjacent digits is always zero. Because $\ell \approx m$, (1) with k in τNAF requires on average $m/3$ point additions or subtractions and m applications of ϕ.

3.4 Window Methods

If enough storage space is available, point multiplication can be sped up with window methods which involve precomputations with P and process w bits of k at a time. We consider window methods only on Koblitz curves in order to keep discussion focused, although analogous algorithms exist also for general curves.

Solinas presented an algorithm for producing *width-w τNAF* in [26]. Instead of using that algorithm, we use the τNAF algorithm which is simpler to implement in hardware and interpret its results as width-w τNAF by replacing certain strings of 0, 1, and $\bar{1}$'s with window values. The resulting representation has an average weight of $H(k) = \ell/(w + 1)$.

We use $w = 4$ and replace $10\bar{1}$ by 3, $\bar{1}01$ by $\bar{3}$, 101 by 5, $\bar{1}0\bar{1}$ by $\bar{5}$, $\bar{1}00\bar{1}$ by 7, and 1001 by $\bar{7}$. For instance, the width-4 τNAF for the τNAF $10\bar{1}010001001$ is $3010000007\bar{}$, and $H(k)$ has reduced from 5 to 3. Precomputed points, P_3, P_5, and P_7, are computed as $P_3 = \phi^2(P) - P$, etc [26].

PLA signature generation requires computation of a point multiplication [3] which is computed efficiently with window methods.

An algorithm for width-w window point multiplication on Koblitz curves is given in Alg. 1.

3.5 Multiple Point Multiplication

Several algorithms require computation of sums of two or more elliptic curve point multiplications. A sum of n elliptic curve point multiplications is called *multiple point multiplication* and it is defined by

$$Q = \sum_{i=1}^{n} k^{(i)} P^{(i)} \tag{6}$$

where $k^{(i)}P^{(i)}$ are point multiplications as defined by (1).

Naturally, (6) can be computed with n applications of (1) and $n - 1$ point additions combining them. However, all n point multiplications can be computed simultaneously with the so-called *Shamir's trick* [10]. Consider the case $n = 2$. The integers are represented as a table with $k^{(1)}$ and $k^{(2)}$ as rows. First, $P^{(1)} + P^{(2)}$ is precomputed. Analogously with the double-and-add algorithm, point multiplication proceeds column by column so that $P^{(1)}$ is added if the column is $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, the point $P^{(2)}$ if $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$, and the precomputed point if $\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]$. Generalization of Shamir's trick for n point multiplications is straightforward but requires more precomputations.

Because zero columns do not require point additions, it is possible to reduce computational cost by representing $k^{(i)}$ with signed-binary representations and choosing the representations which maximize the number of zero columns. A representation maximizing the number of zero columns is called *τ-adic joint sparse form* (τJSF). An algorithm for computing τJSF for $n = 2$ was presented in [8] and it was generalized for $n > 2$ in [4]. Table 1 lists Hamming weights for representations relevant in this paper. They were computed from probabilities given in [4]. PLA signature verification requires a 3-term point multiplication [3].

Table 1. Average Hamming weights $H(k)$ for n 163-bit integers with different representations

n	Binary	τNAF	Width-4 τNAF	τJSF
1	81.50	54.33	32.60	54.33
2	122.25	90.56	—	81.50
3	142.63	114.70	—	96.13

An algorithm for n-term multiple point multiplication on Koblitz curves is given in Alg. 2.

4 Architecture of the Accelerator

The objective for the accelerator is to provide very high throughput while maintaining low computation times for single operations. Because the motivation for designing the accelerator arises from the PLA, it must support both 1-term and 3-term (multiple) point multiplications which are used in signing and verifying packets, respectively. 2-term point multiplications are naturally supported too because they are a special case of 3-term point multiplications where $k^{(3)} = 0$. Algs. 1 and 2 share a common structure, i.e. both require conversions for integer(s) and precomputations, have the same for-loop, and convert the point Q back to $\mathcal{A}$ in the end. We utilize this common structure in implementing the algorithms and use operation-specific processing units in order to increase efficiency.

A simplified top-level view of the accelerator is presented in Fig. 1. The accelerator operates as follows. First,

Algorithm 1 Window algorithm

Input: Integer k, point P
Output: Result point $Q = kP$
  $\langle k_{\ell-1}k_{\ell-2}\dots k_1k_0\rangle \leftarrow w\text{-}\tau\text{NAF}(k)$
  $P_1, P_3, \dots, P_{2^{w-1}-1} \leftarrow \text{Precompute}(P)$
  $Q \leftarrow \mathcal{O}$
  for $i = \ell - 1$ **down to** 0 **do**
    $Q \leftarrow \phi(Q)$
    if $k_i \neq 0$ **then**
      $Q \leftarrow Q + \text{sign}(k_i)P_{|k_i|}$
    end if
  end for
  $Q \leftarrow \text{xy}(Q)$

Algorithm 2 Multiple point multiplication algorithm

Input: n integers $k^{(1)}, \dots, k^{(n)}$, n points $P^{(1)}, \dots, P^{(n)}$
Output: Result point $Q = \sum_{i=1}^{n} k^{(i)}P^{(i)}$
  $\langle k_{\ell-1}k_{\ell-2}\dots k_1k_0\rangle \leftarrow \tau\text{JSF}(k^{(1)}, \dots, k^{(n)})$
  $P_1, P_2, \dots, P_{\frac{3^n-1}{2}} \leftarrow \text{Precompute}(P^{(1)}, \dots, P^{(n)})$
  $Q \leftarrow \mathcal{O}$
  for $i = \ell - 1$ **down to** 0 **do**
    $Q \leftarrow \phi(Q)$
    if $k_i \neq 0$ **then**
      $Q \leftarrow Q + \text{sign}(k_i)P_{|k_i|}$
    end if
  end for
  $Q \leftarrow \text{xy}(Q)$

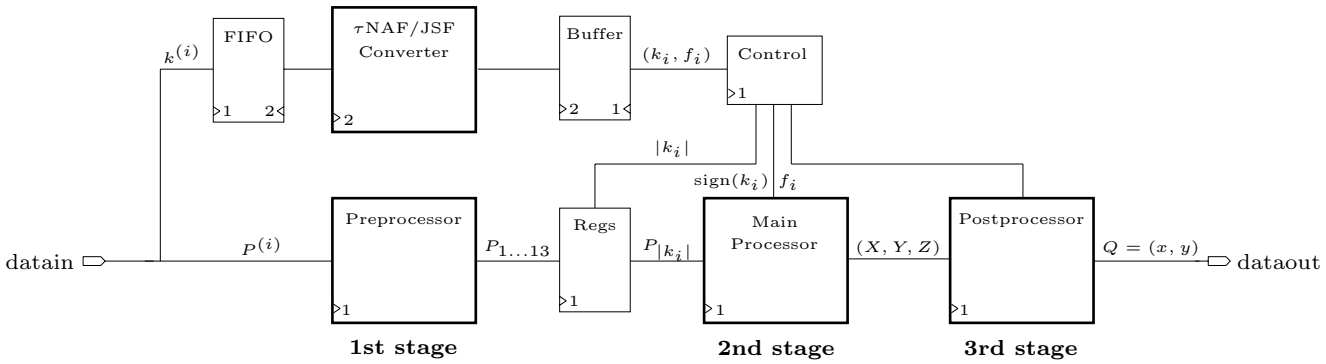

Figure 1. Top-level view of the accelerator. The accelerator includes four main components: τNAF/JSF converter, preprocessor, main processor, and postprocessor. The accelerator uses two clocks which are differentiated with a number in the figure. Operations form a pipeline with three stages as depicted in the bottom of the figure.

integer(s) $k^{(i)}$ and point(s) $P^{(i)}$ are sent to the *τNAF/JSF converter* and *preprocessor*, respectively. The converter converts integer(s) to either width-4 τNAF or τJSF and saves the result into a buffer. Simultaneously, the preprocessor performs precomputations and stores points into the registers. When both converter and preprocessor are ready, the *main processor* executes the for-loop of Algs. 1 and 2. The *control logic* selects one of the precomputed points according to the current k_i and the main processor adds or subtracts it to or from a temporary value $Q = (X, Y, Z)$ and performs the following f_i Frobenius endomorphisms. When the for-loop has been executed, the control logic enables the *postprocessor* which computes the affine representation of the point Q.

The accelerator comprises a three-stage pipeline and is capable of processing three different (multiple) point multiplications at a time. The converter and the preprocessor form the first stage, the main processor the second stage, and the postprocessor is the third stage as depicted in Fig. 1.

4.1 τNAF/JSF Converter

The τNAF/JSF converter consists of five subcomponents as depicted in Fig. 2. It supports computation of three representations: width-4 τNAF, 2-term τJSF, and 3-term τJSF. The 2-term and 3-term τJSF conversions are essentially the same operation, but $k^{(3)}$ is set to 0 in the 2-term τJSF.

The converter first converts all integers into τNAF simultaneously with three τNAF converters which are implemented as presented in [14]. In the width-4 τNAF and 2-term τJSF conversions one or two τNAF converters are idle. Parallel processing was used in order to minimize total computation time.

The converters output τNAF representations one signed-bit per clock cycle starting from the lsb. The signed-bits are fed into three 4-signed-bit shift registers which are used in producing width-4 τNAF and τJSF representations. The conversion to width-4 τNAF from τNAF is trivial. The values of the shift registers are simply replaced by width-4 values as discussed in Sec. 3.4. The conversion to τJSF is slightly more complex. The algorithm from [4] is implemented so that the values of the shift registers are input into a circuit which determines whether the values are reducible or not. If they are and there are no all-zero columns, then the shift registers are updated with reduced values.

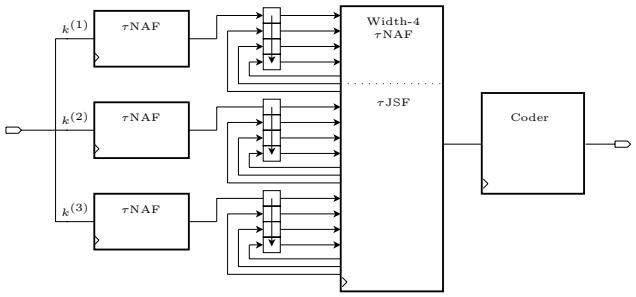

Figure 2. τNAF/JSF converter.

Finally, a width-4 τNAF or τJSF representation is coded into a form that is easily interpreted by the main processor. The code represents the original representation requiring approximately m symbols with $H(k)$ symbols. The code uses symbols (k_i, f_i) where k_i is a nonzero and f_i is the number of following zeros plus one. However for the lsb symbol, f_i is only the number of zeros. For instance, $\bar{1}003000\bar{5}00$ is coded as $(\bar{1}, 3)(3, 4)(\bar{5}, 2)$. When used starting from the msb symbol, each symbol interprets so that k_i gives the precomputed point used in point addition and f_i

gives the number of Frobenius endomorphisms following that point addition. Because the τNAF/JSF converter outputs the code starting from the lsb but the main processor operates starting from the msb, the code is read from the buffer in an inverter order.

4.2 Preprocessor

The preprocessor is based on the architecture presented in [17], but it uses polynomial basis instead of normal basis. Fig. 3 depicts the preprocessor. The storage RAM is used for storing temporary variables during precomputations and it is implemented with embedded memory. The preprocessor is controlled by a finite state machine and hand-optimized microcode which is stored in embedded memory.

Points that are precomputed in the preprocessor are listed in Table 2. The points are represented in $\mathcal{A}$ so that (3) can be used in the for-loop. Precomputatations utilize unified point addition and subtraction formulae which compute both P_1+P_2 and P_1-P_2 with only one inversion [15]. The number of inversions are further reduced by using the so-called *Montgomery's trick* which trades inversions to multiplications (see [15], for example). The precomputations for $n=3$ are computed with Alg. 1 from [15]. If $n=2$, only two points need to be precomputed.

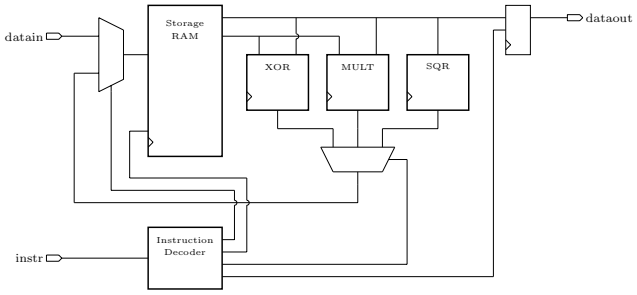

Figure 3. Preprocessor.

4.3 Main Processor

The main processor implements the for-loop of Algs. 1 and 2. This for-loop dominates the computational requirements and, hence, its efficient computation is necessary. The sequential nature of the loop forbids computing point operations in parallel. Parallelism can be used in Frobenius endomorphisms (squarings for all coordinates computed in parallel) and point additions. However, data dependencies usually prevent efficient use of parallelism in point additions and cause poor latency-area products. For example, (3) has critical paths of 8, 5, and 4 multiplications with one, two, and three multipliers, respectively, and further reductions are not possible [17]. Hence, three multipliers must be used for halving the number of multiplications on the critical path thus degrading latency-area product by 50 %.

The following method circumvents this problem in the case of Koblitz curves and point additions using (3). Details of the method are available in an at the moment unpublished article [16]. The method computes consecutive point additions and Frobenius endomorphisms efficiently with parallel field multipliers by interleaving successive operations [16]. The key observation is that the computation of Z_3 in (3) does not require Y_1. Hence, the Z coordinate of the next point addition can be computed simultaneously with the Y coordinate of the previous point addition. This allows reducing the effective critical path to only 2 multiplications with four multipliers without degrading the latency-area product [16].

The method redefines (3) so that the computation is performed with eight subcomputations each including one multiplication. They are defined as follows [16]:

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

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

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

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

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

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

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

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

The subcomputations allow successive point additions with effective critical paths of 4, 3, or 2 multiplications per point addition with two, three, or four multipliers, respectively [16]. We use four multipliers so that one of them is devoted for the Z coordinate computations, (7)–(8), one for

Table 2. Precomputed points

Point	Operation	Point	Operation
Window, $w=4$			
P_1	P	P_5	$\phi^2(P)+P$
P_3	$\phi^2(P)-P$	P_7	$-\phi^3(P)-P$
Multiple, $n=2,3$			
P_1	$P^{(1)}$	P_8	P_3+P_2 †
P_2	$P^{(2)}$	P_9	P_3-P_2 †
P_3	$P^{(3)}$ †	P_{10}	P_8+P_1 †
P_4	P_2+P_1	P_{11}	P_8-P_1 †
P_5	P_2-P_1	P_{12}	P_9+P_1 †
P_6	P_3+P_1 †	P_{13}	P_9-P_1 †
P_7	P_3-P_1 †		

† Computed only if $n=3$

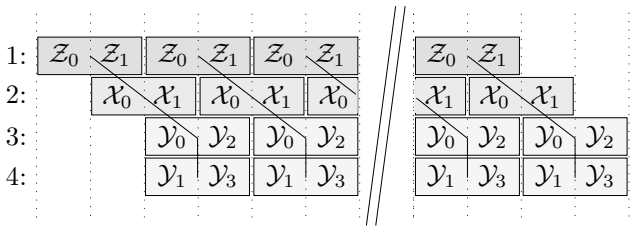

Figure 4. Computation schedule of the main processor [16]. Operations connected with a line belong to the same point addition.

the X coordinate computations, (9)–(10), and two for the Y coordinate computations, (11)–(14). The computation schedule, given in Fig. 4, clearly shows that the effective critical path is 2 multiplications per point addition.

Because each multiplier is used for only two subcomputations, specialized processing units optimized for these subcomputations were designed. The processing units consist of a multiplier and several adders and squarers as shown in Fig. 5. The processing units compute the subcomputations so that their latency is the latency of multiplication plus one clock cycle. When a result coordinate is ready (after $\mathcal{Z}_1$, $\mathcal{X}_1$, and $\mathcal{Y}_{2/3}$), a squarer is used for computing Frobenius endomorphisms for that coordinate. Each Frobenius endomorphism requires one clock cycle. This architecture is presented in more detail in [16].

The main processor does not differentiate between width-4 τNAF, 2-term τJSF, and 3-term τJSF computations. It simply adds or subtracts a precomputed point selected by the control logic to or from Q and performs the following f_i Frobenius endomorphisms.

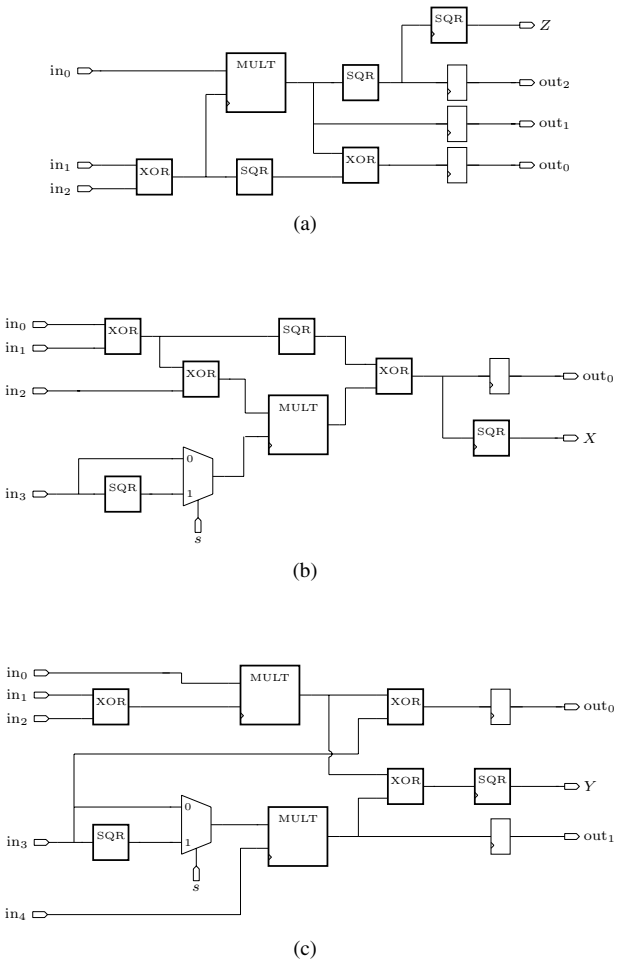

Figure 5. Processing units for (a) $\mathcal{Z}_0$ and $\mathcal{Z}_1$, (b) $\mathcal{X}_0$ and $\mathcal{X}_1$, and (c) $\mathcal{Y}_0$, $\mathcal{Y}_1$, $\mathcal{Y}_2$, and $\mathcal{Y}_3$.

4.4 Postprocessor

The postprocessor maps the output of the main processor from $\mathcal{LD}$ to $\mathcal{A}$, i.e. it computes (2), the last line of Algs. 1 and 2. The computation requires one inversion, one squaring, and two multiplications, of which the inversion is the most complex operation by far. As discussed in Sec. 3.1, inversions can be computed with the Extended Euclidean Algorithm or Fermat's Little Theorem. We selected Fermat's Little Theorem as suggested by Itoh and Tsujii [13] because it uses successive squarings and multiplications which allows reusing the same hardware for inversion and other operations required by (2). The architecture of the postprocessor was presented in [16] and it is depicted in Fig. 6.

The computation of (2) requires 11 multiplications and 163 squarings in $\mathbb{F}_{2^{163}}$ because an inversion in $\mathbb{F}_{2^m}$ is computed with $\lfloor \log_2(m-1) \rfloor + H(m-1) - 1$ multiplications and $m-1$ squarings [13].

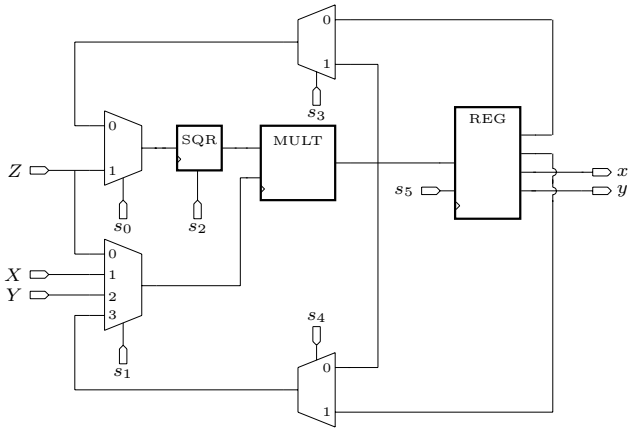

Figure 6. Postprocessor.

5 Analysis and Optimizations

The latencies of operations needed in Algs. 1 and 2 are listed in Table 3. The latencies of precomputations, fop-loop, and coordinate conversion depend on the latency of multiplication in $\mathbb{F}_{2^m}$ denoted by M. Digit-serial multipliers are used as discussed in Sec. 3.1 and their latencies are given by

$$M = \lceil m/D \rceil + 1 \ . \tag{15}$$

The digit size, D, defines both the latency and the size of a multiplier. Because of the round up in (15), only certain values of D are feasible, and other values only add area without decreasing latency. Different D can be used for multipliers in different parts of the accelerator with the exception that all multipliers of the main processor must have the same D.

The latencies of the preprocessor, the main processor, and the postprocessor are plotted as functions of D in Fig. 7. The figure also plots the normalized latency of the τNAF/JSF converter ($185/85 \cdot 499 \approx 1086$) assuming clock frequencies of 85 MHz and 185 MHz for the converter and the rest of the accelerator, respectively; see Fig. 1. If fast multipliers (large D) are used, the constant latency of the τNAF/JSF converter becomes a bottleneck. In order to avoid this, we optimized the accelerator for 1-term point multiplication by choosing D so that τNAF/JSF conversions are computed slightly faster than the other operations of window point multiplications. Hence, we selected $D = 4$ for the preprocessor, $D = 13$ for the main processor, and $D = 3$ for the postprocessor. The postprocessor was selected to be faster than the τNAF/JSF converter in order to ensure that inversions do not become the bottleneck.

6 Results and Comparisons

The architecture was described in VHDL and synthesized for Stratix II EP2S180F1020C3 [2] with Quartus II 6.0 SP1 design software. The synthesis was constrained by clocks of 85 MHz and 185 MHz for the τNAF/JSF converter and the rest of the accelerator, respectively, and the constraints were met.

Table 3. Latencies

Operation	Latency (clock cycles)
Conversion, w-τNAF	499
Conversion, τJSF	499
Precomputation, $w = 4$	$18M + 314$
Precomputation, $n = 2$	$13M + 338$
Precomputation, $n = 3$	$46M + 665$
For-loop	$2H(k)(M + 1) + \ell + 6$
Affine coordinates	$11M + 175$

Table 4. Area consumptions

Component	ALUTs	Regs.	ALMs	M4Ks
Converter	4,906	2,862	2,862	7
Preprocessor	2,037	1,546	1,332	14
Main processor	16,642	10,045	10,930	0
Postprocessor	2,874	2,336	1,953	0
Total	26,616	16,966	16,930	21

Table 4 presents the area consumption of the accelerator and its components as given by the design software. The main processor expectedly dominates in the area consumption. Notice that the total value is not a sum of components because the toplevel includes interface logic and some ALMs share parts from different components. Stratix II S180C3 includes 143,520 ALUTs in 71,760 ALMs and 768 M4K memory blocks [2]. Hence, the accelerator occupies only 23.6 % of the device resources (ALMs) and four parallel accelerators would fit into a Stratix II S180.

Computation times and maximum throughputs are presented in Table 5. The computation time is the time in which the accelerator computes a single operation with an average $H(k)$ when the pipeline is empty, i.e. when no wait delays occur. In all cases throughput is bounded by the main processor. Because four accelerators would fit into a Stratix II S180, we estimate that throughputs of 645 kops, 283 kops, and 242 kops are achievable with one device for $n = 1$, $n = 2$, and $n = 3$, respectively. Notice, however, that these values are highly approximative because we neglect possible decreases in maximum clock frequencies that may occur when place&route becomes more difficult.

Table 6 presents a comparison to other publications. The τ-adic column indicates whether the implementation includes a τ-adic converter. Our accelerator is clearly faster than most published implementations, although comparisons are difficult between different FPGAs. However, Stratix II, Virtex-E, and Virtex-II are roughly at the same performance levels (see [16], for example). The closest counterparts of our accelerator are our previous works presented in [15, 16]. Higher throughput for 3-term point multiplications is achieved in [15], but our accelerator occupies only one-fourth of the area compared to [15]. If four parallel accelerators were used, our accelerator would achieve

Table 5. Performance with clocks of 85 MHz and 185 MHz in a Stratix II S180C3

Operation	Time (μs)	Throughput (ops)
Alg. 1, $w = 4$	16.36	161,290
Alg. 2, $n = 2$	24.28	70,773
Alg. 2, $n = 3$	35.06	60,603

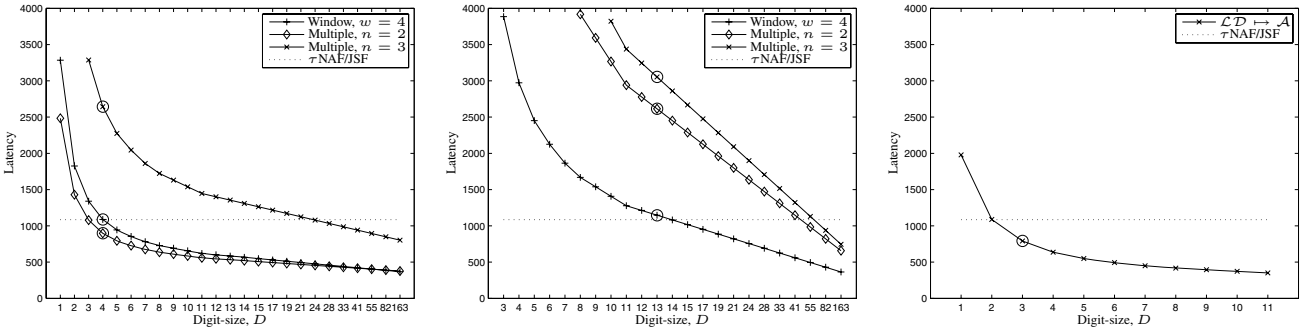

Figure 7. Latencies as functions of the multiplier digit size, D, for (a) the preprocessor, (b) the main processor, and (c) the postprocessor. Values of $D = 4$, $D = 13$, and $D = 3$ were selected for the implementation and the corresponding latencies are circled in the figure.

Table 6. Comparison of published FPGA-based implementations using Koblitz curve, NIST K-163

Ref.	n	Device	Area	τ-adic	Time (μs)	Throughput (ops)
Dimitrov [9]	1	Virtex-II	6,494 slices, 6 B-RAMs	Yes	35.75	27,972
Järvinen [15]	3	Stratix II	67,467 ALMs, 240 M512s, 305 M4Ks	Yes	114.2	166,000
Järvinen [16]	1	Stratix II	26,148 ALMs	No	4.91	203,670
Järvinen [17]	1	Stratix II	13,472 ALMs, several M512s and M4Ks	Yes	25.81	49,318
Lutz [21]	1	Virtex-E	10,017 LUTs, 1,930 FFs	No	75	13,333
Okada [25]	1	Flex 10K	——	No	45600	22
This work	1	Stratix II	16,930 ALMs, 21 M4Ks	Yes	16.36	161,290
This work	3	Stratix II	16,930 ALMs, 21 M4Ks	Yes	35.06	60,603

242,000 ops which outperforms [15] by 46 %. Comparisons to [16] are unnecessary because our accelerator includes the same architecture as a subcomponent (the main processor).

The superiority of Koblitz curves over general curves is evident. The fastest general curve implementation using the same field size ($m = 163$) was recently presented in [7] and it achieves computation time 19.55 μs and throughput 51,120 ops in a Virtex-4 FPGA. Both of these values are inferior to our accelerator, even though they were achieved in a newer and faster FPGA.

7 Conclusions and Future Work

We described an FPGA-based accelerator for elliptic curve operations on NIST K-163 Koblitz curve. The accelerator utilizes window methods for point multiplications and supports computation of sums of up to three point multiplications. The accelerator is capable of computing up to 161,290 ops while still maintaining short computation times. This was achieved because the accelerator uses dedicated processing units for different parts of the algorithms and supports pipelined computation of up to three simultaneous (multiple) point multiplications. The use of dedicated processing units for precomputations (the preprocessor), for-loop (the main processor), and the coordinate conversion (the postprocessor) was shown to offer considerable performance increases with only minor area costs. An important feature that improved speed-area efficiency was the use of six small multipliers instead of one large multiplier as in most publications. To the best of our knowledge, the accelerator outperforms all previously published implementations in computation time and throughput. As expected, Koblitz curves proved to be faster than general curves.

The results demonstrate that modern FPGAs contain enough resources and are fast enough to implement very high performance public-key cryptosystems. Even very demanding communication schemes, such as the PLA, could be feasible in practice if they were accelerated with FPGAs.

We will continue the development of our accelerator. An obvious problem is the fact that τ-adic conversions easily become a bottleneck, as stated already in [16]. Other parts of the circuitry can be made faster at the expense of using more area, but the converter architecture of [14] does not scale up easily. Hence, we are currently searching for both faster and more scalable architectures for the τ-adic conversion. This paper discussed only one specific Koblitz curve, but standards include Koblitz curves over other field sizes as well; see [6, 24], for example. The main processor architec-

ture scales up easily to larger field sizes [16] and, hence, it is likely that our accelerator performs well also with larger field sizes. Nevertheless, this will be studied in the future. In this paper, we focused on optimizing a single accelerator unit, but if several accelerators are operating in parallel, it is likely that higher throughput is achieved by optimizing the architecture similarly as in [15]. Some performance increases could also occur by using combings, larger windows, e.g. $w = 5$, or by introducing more precomputations when $n = 2$, i.e. by using width-w τNAFs for producing a variation of τJSF.

References

[1] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman. A new addition formula for elliptic curves over $GF(2^n)$. *IEEE Trans. Comput.*, 51(8):972–975, Aug. 2002.

[2] Altera. Stratix II device handbook. Datasheet, vol. 1–2, ver. 4.1, Apr. 2006.

[3] B. B. Brumley. Efficient three-term simultaneous elliptic scalar multiplication with applications. In *Proc. 11th Nordic Workshop Secure IT Systems, NordSec 2006*, pages 105–116, Linköping, Sweden, Oct. 19–20, 2006.

[4] B. B. Brumley. Left-to-right signed-bit τ-adic representations of n integers. In *Proc. 8th Int. Conf. Information and Communications Security, ICICS 2006*, volume 4307 of *Lecture Notes in Comput. Sci.*, pages 469–478. Springer, 2006.

[5] C. Candolin, J. Lundberg, and H. Kari. Packet level authentication in military networks. In *Proc. 6th Australian Information Warfare & IT Security Conf.*, Geelong, Australia, Nov. 2005.

[6] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography*, Sept. 20, 2000.

[7] W. N. Chelton and M. Benaissa. Fast elliptic curve cryptography on FPGA. *IEEE Trans. VLSI Syst.*, 16(2):198–205, Feb. 2008.

[8] M. Ciet, T. Lange, F. Sica, and J.-J. Quisquater. Improved algorithms for efficient arithmetic on elliptic curves using fast endomorphisms. In *Advances in Cryptology, EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Comput. Sci.*, pages 388–400. Springer, 2003.

[9] V. S. Dimitrov, K. U. Järvinen, M. J. Jacobson, W. F. Chan, and Z. Huang. FPGA implementation of point multiplication on Koblitz curves using Kleinian integers. In *Cryptographic Hardware and Embedded Systems, CHES 2006*, volume 4249 of *Lecture Notes in Comput. Sci.*, pages 445–459. Springer, 2006.

[10] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, 31(4):469–472, July 1985.

[11] J. Goodman and A. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE J. Solid-State Circuits*, 36(11):1808–1820, Nov. 2001.

[12] M. A. Hasan. Look-up table-based large finite field multiplication in memory constrained cryptosystems. *IEEE Trans. Comput.*, 49(7):749–758, July 2000.

[13] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inform. Comput.*, 78(3):171–177, Sept. 1988.

[14] K. Järvinen, J. Forsten, and J. Skyttä. Efficient circuitry for computing τ-adic non-adjacent form. In *Proc. 13th IEEE Int. Conf. Electronics, Circuits and Systems, ICECS 2006*, pages 232–235, Nice, France, Dec. 10–13, 2006.

[15] K. Järvinen, J. Forsten, and J. Skyttä. FPGA design of self-certified signature verification on Koblitz curves. In *Cryptographic Hardware and Embedded Systems, CHES 2007*, volume 4727 of *Lecture Notes in Comput. Sci.*, pages 256–271. Springer, 2007.

[16] K. Järvinen and J. Skyttä. Fast point multiplication on Koblitz curves: Parallelization method and implementations. *Microproc. Microsyst.* submitted.

[17] K. Järvinen and J. Skyttä. On parallelization of high-speed processors for elliptic curve cryptography. *IEEE Trans. VLSI Syst.* in press.

[18] N. Koblitz. Elliptic curve cryptosystems. *Math. Comput.*, 48:203–209, 1987.

[19] N. Koblitz. CM-curves with good cryptographic properties. In *Advances in Cryptology, CRYPTO '91*, volume 576 of *Lecture Notes in Comput. Sci.*, pages 279–287. Springer, 1991.

[20] J. López and R. Dahab. Improved algorithms for elliptic curve arithmetic in $GF(2^n)$. In *Selected Areas in Cryptography, SAC'98*, volume 1556 of *Lecture Notes in Comput. Sci.*, pages 201–212. Springer, 1999.

[21] J. Lutz and A. Hasan. High performance FPGA based elliptic curve cryptographic co-processor. In *Proc. Int. Conf. Information Technology: Coding and Computing, ITCC 2004*, volume 2, pages 486–492, Las Vegas, NV, USA, Apr. 5–7, 2004.

[22] G. Meurice de Dormale and J.-J. Quisquater. High-speed hardware implementations of elliptic curve cryptography: A survey. *J. Syst. Architect.*, 53(2-3):72–84, Feb.-Mar. 2007.

[23] V. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology, CRYPTO 1985*, volume 218 of *Lecture Notes in Comput. Sci.*, pages 417–426. Springer, 1986.

[24] National Institute of Standards and Technology (NIST). Digital signature standard (DSS). Federal Information Processing Standard, FIPS PUB 186-2, Jan. 27, 2000.

[25] S. Okada, N. Torii, K. Itoh, and M. Takenaka. Implementation of elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA. In *Cryptographic Hardware and Embedded Systems, CHES 2000*, volume 1965 of *Lecture Notes in Comput. Sci.*, pages 25–40. Springer, 2000.

[26] J. A. Solinas. Efficient arithmetic on Koblitz curves. *Des. Codes Cryptography*, 19(2–3):195–249, 2000.

[27] T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Trans. Embed. Comput. Syst.*, 3(3):534–574, Aug. 2004.